Smart Contract for Certificates of Authenticity for Luxury Fashion Items

“ProVeReal”

Supervised By Nikos Tzevelekos
Undertaken By Kiran Fernando
k.fernando@se16.qmul.ac.uk, +447772601347

Queen Mary University of London

Submission Date : 24/04/19

School of Electronic Engineering and Computer Science

Acknowledgements

I would like to thank my parents, Jay Fernando and Hardarshan Kaur for all their support and guidance; my coursemate Mehdi Dahmani for helping me understand various topics throughout the course of my degree; my friends Raymond Poon, Levent Akyildiz, John Punzal, Adarsh Das, Vincent Mak, Ashtak Maharaj, Danial Naqvi and the members of the “ITLSquad” group chat.

Especially, I would like to thank my supervisor Nikos Tzevelekos for helping to structure my report, overcome key issues and kindly giving me the opportunity to discuss my progress on a fornightly basis.

Disclaimer

Contents

Executive Summary **3**
- Problem 3
- Proposal 4
- Implementation 4
- Testing 4
- Further Work 4

Background **5**
- Counterfeiting of Fashion (Business Case) 5
- Anti-Counterfeiting Measures Currently Used 6
- Blockchain Technology 8
- Smart Contracts and Ethereum 9
- Non Fungible Tokens 11

Project Aims **12**

Design Considerations **12**
- ERC721 Standard 12
- Metadata 13
- Decentralised Storage 15
  - Bittorrent 15
  - Interplanetary File System 15
- Brand Verification 15
- Pricing and Revenue Collection 16
- Contract Development & Deployment 16
- Non Ethereum EVM Blockchains 17
  - Test Networks 17
  - Alternative Main Networks 17
- Interfacing with Smart Contracts 18
  - Web3 18
  - Existing Wallets for ERC-721 Tokens 18

Implementation **19**
- Functionality 20
- Smart Contract 20

Issuance Tool 22
Wallet Functionality via OpenSea 23

Testing 25
Overview 25
Smart Contract Unit Tests 26
Pinata & IPFS gateway Integration Tests 27
Discussion 28
System Tests 29
Gas Cost Discrepancies 30

Further Work 33
Cost optimisation 33
Sidechains 33
Reducing computational intensivity 33
Privacy 33

Conclusion 33

References 33

AppendixA: Truffle Test Table 37

AppendixB: Truffle Test Screenshot 44

AppendixC: Pinata Integration Test Table 45

AppendixD:Pinata Integration Test Screenshot 47

AppendixE: System Test Table 48

Appendix F: Gas Cost Table 51

Appendix G: Gas Ratio Box Plot dataset 52

Executive Summary

Counterfeiting loses the fashion industry $450 Billion every year (The Fashion Law, 2018). The solution presented in this paper “ProveReal” is built around a smart contract designed to run on Ethereum to enable brands to link digital certificates of authenticity to every item they sell which can be verified by the owner of the clothing using a standard Ethereum Decentralised App Explorer, such as the Coinbase Wallet.

Problem

The luxury fashion brands, particularly in the streetwear niche, have very little protect their designs from counterfeiting. With streetwear items of limited edition being resold at up to 1200 times their original sale price - there is huge incentive to counterfeit. (Isichei, 2018). This is especially hurtful to consumers who are willing to pay top dollar for genuine goods. Poor quality fakes and a lack of awareness for genuine clothes also damages a brand's value.

Proposal

This project aims to create a tamper proof platform for brands to issue digital certificates of authenticity and ownership that can be issued to their customers on the blockchain, and can be transferred as the physical fashion items change hands. This will be done using an Ethereum smart contract implementing a non fungible token system, which will be deployed on an Ethereum test network. An important feature will be security features, that only allow entities (ethereum addresses) approved by the contract deployer to mint certificates. Each approved entity will have a brand name attribute. As ProveReal is designed as a revenue making service, the smart contract will also administer fees for certificate issuance - we will use the native token of the network as the currency used to price and levy fees.

Implementation

The smart contract is deployed to the Rinkeby test network, and has features to prevent unauthorised use. The issuance tool, developed using Python, also pins metadata for items to the Interplanetary File System, a decentralised file storage network, to prevent any manipulation of data once a certificate has been issued.

Testing

The contract and issuance tool have been throughly tested using 46 unit, integration and system tests. Flaws exposed in the tests were analysed to find flaws in the testing approach and in the systems used. However, none of these effected the running of the system

Further Work

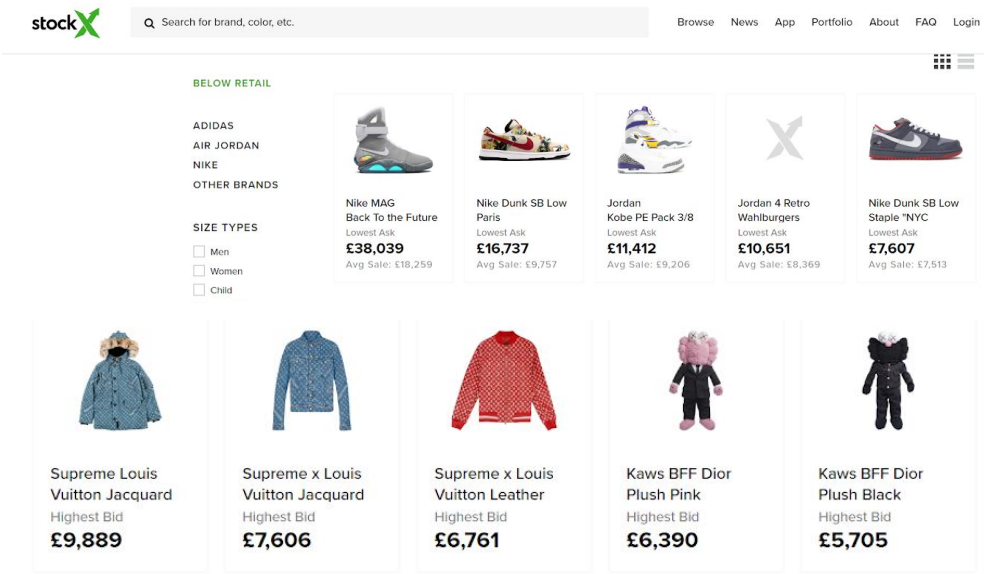
The further work section describes potential steps to improve the effectiveness of system, in particular optimising the contract, increasing privacy and deploying to a faster distributed network.

Background

Counterfeiting of Fashion (Business Case)

Counterfeit fashion is a $450 billion business (The Fashion Law, 2018) operating over the $3 Trillion fashion industry (Strijbos 2016). Consumers are particularly at risk when buying goods from a 3rd party party such as a reseller - which is common in the streetwear space. The streetwear niche accounts for $300 billion (Lea 2018) of the Fashion Industry . In the modern era streetwear has gained notoriety, as a high end and sought after fashion. The streetwear industry is centred upon “hype”-oriented brands, most notably Supreme, that create scarcity not through high prices, like luxury brands, but through limited production. (Helmore, 2018)

Distribution of these items either occurs online, or through “drops” (Helmore, 2018), that happen in brick and mortar locations - with long queues resulting. The limited edition nature of the items (shoes, tshirts and hoodies, for example), results them being sold on the resale market for sizeable markups of up to 1200 times their original sale price (Isichei, 2018).



Above : A selection of highly expensive streetwear items listed on StockX.com

Due to such markups it is often that counterfeiters can go to great expense to create replica items. Streetwear collectors and “sneakerheads” have long relied on build quality, stitching, and attention to detail to determine if wares are from genuine sellers

(Dejolde 2013), however, with the increasing quality of counterfeits, it is becoming difficult for enthusiasts to tell the difference (HBO, 2018).

Anti-Counterfeiting Measures Currently Used

Anti-counterfeiting efforts centring around the targeting resellers of counterfeits include suing retailers and taking down websites. Major streetwear players, such as Off-White™ and Nike, have gone to great lengths to curb the counterfeiting market, such as sueing retailers, and taking down misleading websites (Stanley 2018) (Mantor 2018). However, they have struggled to take down counterfeiters in China - which run rampant.

Amazon's new Project Zero takes this approach to a higher level with the use of artificial intelligence to automatically remove suspect stores from their platform – in partnership with brands. (Amazon 2019)

A simple solution would be to maintain a list of authorised sellers, however, with the business model of streetwear brands being direct to consumer, and there being a high number of sole trader resellers, a system like this would be impractical. The majority of resellers use online platforms such as Depop, Grailed, Ebay and StockX to sell their items, as well as on facebook groups. (Bearne 2017)

Above: StockX Authenticator verifying pair of sneakers. (Mitchell K, 2018)

To ensure the legitimacy of items, StockX and Grailed, two streetwear resale platforms, use the visual and physical inspection of items to determine if items being sold

through them are not counterfeit. StockX employ a large team of authenticators, individuals with acumen in the field of streetwear fashion, to verify the authenticity of goods through physical inspection (Reindl 2018). Whilst this gives consumers confidence, it is costly, given it is labour intensive, and also given the profit motive for fakes driving up quality (HBO, 2018) - fakes can still make it past inspection. Grailed use a team of moderators to check for suspicious items based on appearance and other factors (Grailed,2019)

Luxury and streetwear brand Moncler, implemented a centralised counterfeit protection system dependent on radio frequency identification (RFID) tags in clothing and server infrastructure (Moncler 2016). This approach requires a high setup cost, due to tooling, and development costs to make an app and backend infrastructure to store ownership data. As well as this, Moncler must maintain the servers required to host the data, maintain the software and be compliant with GDPR, as they hold personal information. The process is long for the consumer, as they must manually put in their details to register their product. As well as this, it has been proven possible to replicate Moncler clothes, in spite of protections (Moncler Expert, 2018). This can be done both via the duplication of the appearance of an RFID and cloning of the QR code, and RFID integrated circuit's contents.

Top Left : Moncler RFID Tag (Alvarez,2017). Top Right : Stone Island Certilogo (Stone Island, 2019)

Stone Island has also developed their own solution called Certilogo (Certilogo 2019a), which they offer to other brands. Certilogo works using a similar method to Moncler's solution and faces the same flaws. To prevent counterfeiting of tags, tags get blacklisted when scanned too many times over a given period - this is to prevent mass cloning of items. However, this is of great disadvantage to sellers who rely on being able to prove legitimacy to sell their goods. Certilogo solves this through their seal of authentication feature - where resellers manually register to produce a secondary code

to allow potential customers to view their items. However, this requires additional protections, such AI to determine if users are genuine. (Certilogo 2019b)

Brands sell their goods through a diverse array of channels, including their own online and physical storefronts, as well as via retailers who may use both online and brick and mortar stores to sell goods. A solution like Moncler's or Certilogo's, whilst requiring new infrastructure and being flawed, means that new technology does not need to be integrated into the retail process. Such a task would be costly, given that Ralph Lauren, for example, currently has 1423 retail locations worldwide (Business Wire, 2019)

Many brands, such as Uniqlo and H&M, also have their own RFID tag based systems, however these are based on paper tags that are removed when the clothing is worn, for the purposes of inventory management (AsiaRFID 2017). However, once the item is sold, there is no means of tracking it, for the end consumer – to verify authenticity.

Recently blockchain has been touted as a solution for both the authenticity and supply chain issues the luxury fashion industry faces (The Fashion Law 2018b). Provenance while providing supply chain solutions, provide a blockchain solution similar to certilogo, where a tag can be used to identify an item, through a digital certificate representing it (Provenance 2019). Recently, Louis Vuitton has announced they have also been working on a similar solution in partnership with Microsoft and Consensys (Allison, 2019).

Blockchain Technology

Blockchain gained prominence when Bitcoin gained popularity and hundreds derived first generation cryptocurrencies were developed in the early 2010s. Bitcoin pioneered serveral key innovations in a real world context, listed below.

A blockchain is a data structure that stores a growing list of immutable records, called blocks. Blocks store changes in the state of the system they represent (predominantly currency) and reference the previous block in the chain (through a hash of the referenced blocks contents) - forming a chain, hence the name. Bitcoin uses SHA-256 as its hashing algorithm. (Nakamota, 2009)

This data structure is that enables decentralised Proof of Work (PoW) networks to built around it. Proof of work was theorised as an alternative to one entity, one vote systems, that can be hijacked by Spoofing IP addresses - known as Sybil attack (Douceur J, 2002).

In PoW, blocks are validated via hashing their data in combination with a nonce to produce a hash with a requisite number of zeros (block difficulty). This is done as open

competition, where anyone has the means to produce the next block (provided they have can provide the required computing power). A block reward of new crypto-currency tokens serves as an incentive to mine the network.

A high block difficulty is important as it increases the cost of conducting Sybil attacks. The value of an economic network is driven by the amount of transactions that take place (block'tivity, 2019), which in turn drive up the value of a crypto-currency. Higher economic incentives increases interest from miners, but also the reward from hijacking a system. Bitcoin solves this using a novel system where the block difficulty is adjusted depending on the total hash power in the network. (Nakamoto, 2009)

This means meaning that the state of data is not dependent on a central entity to update it, and there is mitigated risk of a single entity manipulating block to suit their agenda.

Another key innovation alongside the blockchain itself is the unspent transaction output (UXTO) model for storing units of value, used by Bitcoin. These outputs can only be spent by the owner of that output's private key. Using cryptographic identities creates a level of privacy, as individuals don't need to give personal details to use the network. This is essential as the contents of the network is publicly available. The use of UXTO also makes in quick to verify if transactions are correct.

Smart Contracts and Ethereum

Beyond currency, Bitcoin and bitcoin inspired cryptocurrencies lack functionality. However, a limited scripting language allows for basic escrow contracts (Kaiser, 2017), and technologies such as colored coins (Colored Coins, 2018) allowed for different assets to be traded on a host coin's network.

However, these technologies fail to take full advantage that lack of counterparty risk, and decentralisation of the networks they operate on. This led to the development of Ethereum - a platform for building decentralised blockchain powered applications. Applications run on Ethereum in the form of smart contracts, objects that run on Ethereum nodes, that manipulate variables stored within the Ethereum blockchain. Smart contracts, can't be updated in a conventional sense, and behave like legal agreements, stating what interested parties agree to. However, unlike legal contracts, smart contracts can execute the agreed action when the right conditions are met. Ethereum has its own cryptocurrency called Ether. (Ethereum 2019)

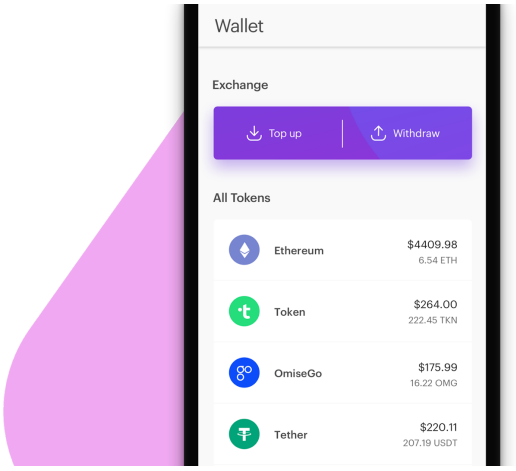
Unlike bitcoin, however, Ethereum does not use the UXTO model and instead operates a a system where public keys correspond to 20byte hexadecimal strings representing accounts. Smart contract also use this same address system, meaning users can

interact with contracts in a similar way to other users - being able to transfer them Ether etc. (Ethereum 2019)

This currency is used to pay for the execution of code in the network, the cost of which is maintained in the Ethereum Yellow Paper (Wood 2019) in units called Gas. Gas is multiplied by the gas price, which varies depending on traffic on the network, to form the cost of execution in Ether (ETH Gas Station, 2018).

Ethereum smart contracts can be developed in an any language that can be compiled into Ethereum Virtual Machine (EVM) code. The most popular language is Solidity, however, others exist, such as : Viper and LLL (Rivlin 2016). Smart contracts take advantage of features of the Ethereum API, such as being able to determine the cryptographic address of a request via the msg.sender (Solidity.readthedocs.io, 2018) property. This interface is what frontend applications leverage to access smart contracts. Smart contracts get their own addresses on the blockchain and get interacted with in a similar way that an address that simply stores Ether does.

Quickly after Ethereum's release, developers realised they could use smart contracts to raise funds and release their own cryptocurrencies, without the hassle of getting miners to mine the network. This has led to Ethereum becoming the number one blockchain platform for initial coin offerings (Draglet 2018).



From organisations looking to replace the US Dollar (MakerDAO, 2018) as the world currency of choice, decentralised funds to P2P loan platforms (Ethlend, 2018) - thousands of smart contracts have been developed.

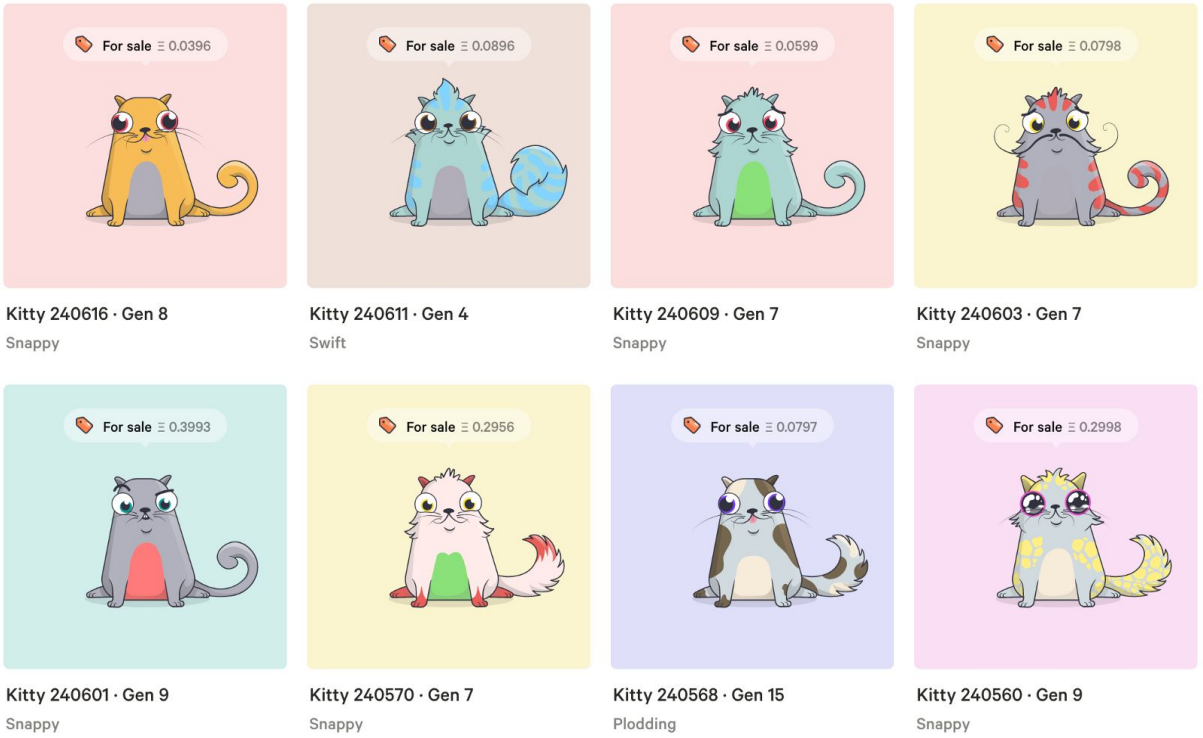
However, as smart contracts can't be updated - vulnerabilities in code can have serious consequences. This was learnt early in the community when The DAO, a decentralised smart contract powered venture capital fund was drained of $70 million worth of Ether in 2016. The event was shocking to the new industry, that it lead to Ethereum being split into two, Ethereum and Ethereum Classic (Falkon, 2017).

Whilst new strategies, to emulate "updating" smart contracts have come into existence (Vergauwen 2018) - the industry mainly focuses on auditing contracts for bugs, before they get deployed to the main network (Grincalaitis, 2017).

Auditing smart contracts is considered professional practice, an example of reputable auditing firm is Zepplin (Zeppelin, 2018). Investors want assurance that code is safe, as do developers themselves. This has lead to the majority of smart contracts being open sourced, with huge libraries of standard contracts being available, such as Zeppelin's Open-Zeppelin (Open-Zeppelin, 2018).

Non Fungible Tokens

As well as financial applications, Ethereum has found use as a platform for holding digital assets. Cryptokitties (Cryptokitties, 2018) is a prime example of this, a game where players collect and breed digital cats, represented as non fungible tokens.



Above : CryptoKitties being sold for Ether via the CryptoKitties Dapp

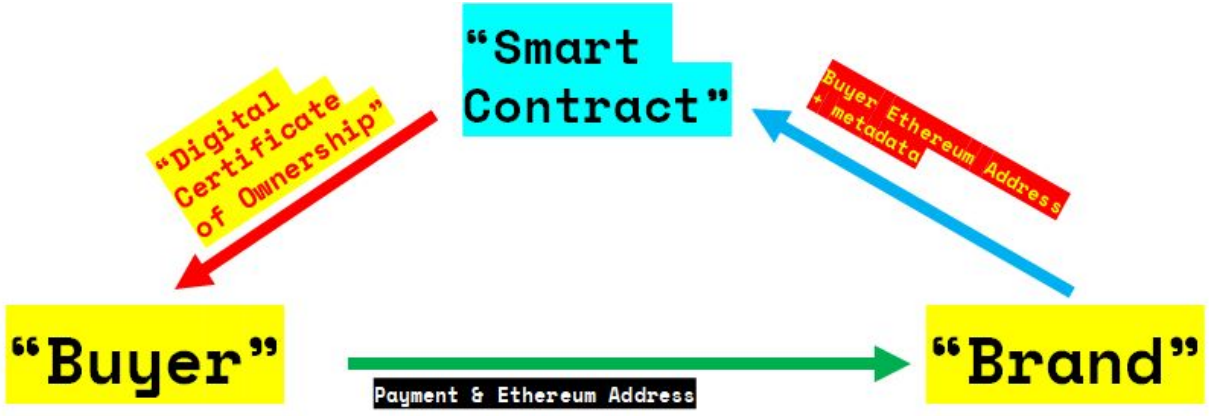
Non fungible tokens are non divisible, distinguishable digital items existing in a blockchain with an owner. This differs from ERC-20, in which all that is stored is balances, and individual units are all identical. These features make them ideal for representing real life assets on the blockchain NFTs are standardised via ERC-721 (Entriken et al, 2018).

Whilst popular for games, companies are looking to use NFT technology to put real objects on the blockchain. Blocksquare, for example, is looking to move land registries to the blockchain (Hamilton, 2018).

Another interesting application of NFT registered assets is in finance, for example, startups Dharma and Centrifuge have demonstrated how Invoices can be used as collateral for loans (Schmitt 2018). This opens up the possibility for other items, such as highly expensive sneakers, to be used as collateral for personal loans. Given how Stock X view themselves as a “Stock Market of Things”, this feature of non fungible tokens could be a very useful tool for the industry as a whole.

Project Aims

The ProveReal System aims to provide a smart contract and issuance tool that aims to give luxury fashion and streetwear brands an interface to leverage the immutablity of blockchain technology to protect themselves and their customers from counterfeiting. This will be done using an ERC721, where each item has a non-fungible token virtually representing it. Below is a diagram showing how the smart contract will be interacted with in a purchase scenario.



Design Considerations

ERC721 Standard

Non-fungible tokens were referred to initially as “deeds” by its inventors. This is comes out strongly in the paper highlighting its design (Entriken 2018). ERC721 describes the interface for a data structure, that can represent and store an infinite number of deeds, and their respective owners and metadata, as well as methods to access and modify this data (create new deeds, transfer deeds, etc). For the purposes of this paper, ERC-721 tokens will be referred to as NFTs, tokens and certificates - as we are using them to make certificates of authenticity. Below is the comment free code for the ERC721 interface, written in Solidity.

```
interface ERC721{
event Transfer(address indexed _from, address indexed _to, uint256 indexed _tokenId);
event Approval(address indexed _owner, address indexed _approved, uint256 indexed _tokenId);
event ApprovalForAll(address indexed _owner, address indexed _operator, bool _approved);
function balanceOf(address _owner) external view returns (uint256);
function ownerOf(uint256 _tokenId) external view returns (address);
function  safeTransferFrom(address _from, address _to, uint256 _tokenId, bytes data) external
payable;
function safeTransferFrom(address _from, address _to, uint256 _tokenId) external payable;
function transferFrom(address _from, address _to, uint256 _tokenId) external payable;
function approve(address _approved, uint256 _tokenId) external payable;
function setApprovalForAll(address _operator, bool _approved) external;
function getApproved(uint256 _tokenId) external view returns (address);
function isApprovedForAll(address _owner, address _operator) external view returns (bool);
}
interface ERC165 {
    function supportsInterface(bytes4 interfaceID) external view returns (bool);
}
```

In the base version above, tokens simply have an ID (unsigned integer), an owner (Ethereum Address), and a list of approved parties that can handle the tokens on the owners behalf. However, whilst a method is defined for getting the owner of a given token (ownerOf), one doesn't exist to find the tokens of a given owner. This is due to the way solidity allows programmers to store and access data. Hashtable like data types called mappings are used to enable the creation of key value tables (Solidity.readthedocs.io 2019). Hash Tables by nature only forward navigable, and due to computations having to be efficient due to block gas limits, constructing more versatile data structures is not possible. Thus the only way to enable two way navigation of related objects is to use two sets of hashtables and keep them consistent.

The optional ERC721EnumerableInterface enables a query of the tokens of an address returning an array of tokenIDs (Entriken 2018). Open-Zeppelin implements this interface in their open source ERC721Enumerable.sol contract through the use of two additional mappings : _ownedTokens (address to array of unsigned integers) to represent the list of tokens owned by a given address and _ownedTokenIndex (unsigned integer to unsigned integer) to represent the position of a given token in its respective _ownedTokens array.

Metadata

Metadata such as item name and size, for example, are essential descriptors of a piece of clothing or footwear. Any NFT implementation would be non-functional if it wasn't possible to derive this data from data held within it.

A simple solution would be to store the data on chain, within the contract itself. An NFT is built around a unique 256bit ID and its relationship to an owner address and metadata (Open Zeppelin, 2018b). To create an itemName field, we could create an mapping (uint256 => string), such that a each UID is can be used as a key to find its name. In solidity the mapping class implements a hashtable (Simon, 2017). However, metadata needs vary from item to item. Sizing formats vary, for example, in men and womens clothes, and accessories, such as bags, do not have sizes at all.

Open Zeppelin already has a reference implementation for an NFT with a built in tokenID to metadata mapping attribute called ERC721Metadata.sol (Open Zeppelin, 2018b). This will be extended appropriately in this project.

Metadata requirements all streetwear items share are: itemName, itemType, Brand and sourceOfPurchase. Size is also a requirement in most clothing, however, as clothing can be sized numerically, using characters (S,M,L etc) and have multiple measurements (e.g. jeans) , there may be some ambiguity. For the purpose of simplicity, the contract will not have size attribute validation, and instead will rely on the frontend client for issuance handling validation.

In addition, an item image field would be ideal - so that applications can display the appearance of the streetwear linked to a given NFT. Storing large files on the blockchain is impractical as this requires roughly 6 million gas per kilobyte There is a limit to the amount of gas that can be processes in a given 15 second Ethereum block - this is currently at roughly 8 million (06/12/18)(Etherscan,2018). So therefore, embedding images directly into the blockchain is impractical. At today's prices (6/12/18), adding a kilobyte of data would cost $6 [40].

Data can be stored off chain, and the NFT can have a reference field for a web address of the metadata it is linked to (Virk, 2018). However, this adds an element of centralisation, which defeats the purpose of using an untamperable blockchain. More concerningly, from a business perspective, running servers creates an additional ongoing cost. This can be addressed by using a decentralised storage system to store item metadata.

A number of ERC-721 wallets (docs.rarebits.io,2019) (docs.opensea.io,2019) have compatibility with metadata URLs - easing the development of a frontend for users. There is also a standard for JSON metadata for ERC-721 that many wallets implement (Nicholas et al 2018) - this standard enables us to attach a picture for the item, an item name, tags, description and key properties of the item. This makes the standard an idea base for display the size, brand, source and type of item.

However, the smart contract would be meaningless if data regarding the creator of an NFT was not present and proveable on the blockchain. Therefore, a balance must be struck between storing data on and off chain. The best way to implement this would be to implement the brand/issuer attribute on chain, and the remaining data off chain.

Interplanetary File System

The interplanetary File System is a distributed network designed to host files, it is popularised in blockchain development due to its self-certified file system, where files IDs are product of hashing their content. This make it easy to verify if content is correctly hosted (Benet 2014). However, due to being a torrent based system, it faces data permenance issues where data must be “pinned” in order to persist in the network. This will be solved by using Pinata, IPFS content hosting service, to pin metadata and images brands will host.

Brand Verification

Without proper verifications it would possible for unauthorised third party to produce seamingly legitmate certificates - as any address could call the contract. To prevent this, we must maintain a list of authorised users (Ethereum addresses). A simple way to do this would be to maintain a mapping of authorised users within our contract (address => bool). A simple if authorisedUser(msg.sender) statement could be used to prevent users outside a permitted list from using the contract. This list could be maintained centrally by ProveReal, where users have to register their brands to be allowed access to the contract.

However, unless further protections are in place, registered entities would be able to counterfeit other registered addresses by making certificates of identical metadata. This could be done through calling an mint (or equivalent method) with the competing brand’s metadataURI as the argument. This could be mitigated by having a second data structure keeping track of which address corresponds to a given brand. This could tied to the given token at the point of issuance - to give the end user a tell tale indication that their item is or is not issued by a given brand.

Alternatively, mapping could keep track of which addresses are allowed to use which metadata, with items having to be registered before use - this would prevent popular items from having their metadata copied.

The Ethereum Name Service (ENS), which is used to provide human readable name resolution to Ethereum addresses and is well supported by Ethereum Wallets, seems like a good choice at first glance. It could potentially empower brands to issue to their consumers without having to Register with ProveReal (they would have to register via

ENS registrar) and create addresses which have use outside the ProveReal platform. However, unfortunately, ENS is insecure for our application as its has no way of enforcing character sets - due to using hashes of strings to map to addresses. This makes in vunerable to international domain style homograph attacks, where deceptive unicode characters (Cyrillic A in place of a Latin A) for example, can be used to trick end consumers. (Ethereum Name Service, 2019) (Zheng X, 2017)

Pricing and Revenue Collection

The ProveReal Contract is designed to generate revenue, in order to do this, it needs mechanisms to set fees for method execution (in Ether), and make it possible to withdraw fees. msg.value (Solidity.readthedocs.io, 2018) is the request parameter used to determine the amount of Ether sent to a contract during a method call, a method call can be thrown using if the value isn't correct, by using the require statement as so: require (msg.value == requiredValue). The given ether would be returned to the sender if this occured.

Due to the volatility of Ethereum (Coin Market Cap, 2018), it would be nonsensical to set the fees as constants, therefore, setters would need to be created to update the fees, to keep it pegged with a fiat currency value.
There would also need to be a withdrawal method, to let the owner of the contract cash out the profit generated by the smart contract.

Both of these actions would need to be secured such that only the owners address can be used to call them. This can be done via extending the Ownable.sol contract in Open Zeppelin, and using the onlyOwner() modifier for the methods (Open Zeppelin, 2018c)

Contract Development & Deployment

Ethereum has a very young ecosystem, and despite the Ethereum network handling billions in value - its development tools are underdeveloped. Additionally the instability of the Ethereum protocol, APIs, EVM languages and the EVM add to the challenge of building smart contracts to run on Ethereum (Breen, 2018).

The workflow for producing a smart contract is such : write a smart contract in an EVM language (ie. Solidity); compile it into EVM bytecode; then deploy this byte code to the blockchain (Davies, 2018)

Contracts are deployed by producing an application binary interface from bytecode, and uploading this data, alongside construction arguments, in a transaction to an

uninitialised address the blockchain (Wesley, 2017). Method calls to the contract will be sent to this address, as this is the contract's reference on the network.

Contracts can be compiled and deployed using the standard tools built by the Ethereum Foundation, namely Solc (the standard solidity compiler) and Web3.js(a means to communicate with the Ethereum network using javascript) (Available via Ethereum's Github). However, more sophisticated tools have been developed.

Truffle (Consensys, 2018a) is good example of a popular Ethereum development tool, it manages contract compilation, linking and migration. It also has a built in system for devising and automatically running unit tests, written in either Solidity or Javascript. It also works in well in conjunction with Ganache, a tool also made by Consensys that allows users to run a local blockchain, which they can use to debug contracts. A strong alternative to Truffle is Embark by Status (Status, 2018). Both work using Node.js, and allow the developer to interface with the contracts via Javascript.

Libraries of pre-audited smart contracts enable developers to focus on building original functionality, as opposed to re-developing common features, and risking the inclusion of a security exploit (Open-Zeppelin, 2018). Currently, the only major offering is from Zeppelin. Open-Zeppelin offers ready to import contract for tokens and permissions management - highly important aspects of this project. These contracts can be imported using an import statement and can be inherited by the contract(s) that will be developed for the scope of this project.

Non Ethereum EVM Blockchains

Test Networks

Due to Ether having a financial cost, developers opt to test their code using test networks, these networks are run by volunteer nodes, who give test tokens to developers to pursue testing. Ropsten, Kovan and Rinkeby are popular test networks. For this project, Rinkeby was chosen to deploy the smart contract.

Alternative Main Networks

In addition to the Ethereum network, there are several other blockchain platforms that support the Ethereum Protocol, most notably Ethereum Classic. Additionally, others such as POA Network (POA Network, 2018) and Thundercore (Thundercore, 2018) have also been developed.

These networks have benefits compared to Ethereum in its current state, for example, Thundercore promises to be capable of handling 1200 transactions per second, much

higher than Ethereum's 13 tps (GoChain, 2018). Being able to handle a high load would be vital for an decentralised application aiming for mainstream use. There is also a cost concern, due to lower gas costs, using Ethereum classic is much cheaper than using the more popular Ethereum network (Gastracker.io, 2018) (ETH Gas Station, 2018).

Cost and speed are two major concerns holding the Ethereum Network back from mainstream use. However, solutions, such as sharding (splitting the network in separate clusters) and sidechains (blockchains that link to Ethereum) are being discussed. (Consensys 2018b)

Interfacing with Smart Contracts

Web3

Smart contracts exist on the blockchain and must be communicated with using an client interface. This interface is Web3.js, Web3.js (Web3js.readthedocs.io, 2018) is an API that enables clients to interact with Ethereum nodes - it enables the development of web based front ends which can leverage smart contracts. It is also used in development frameworks such as Truffle to interface with contracts via the terminal (Quintal 2017).

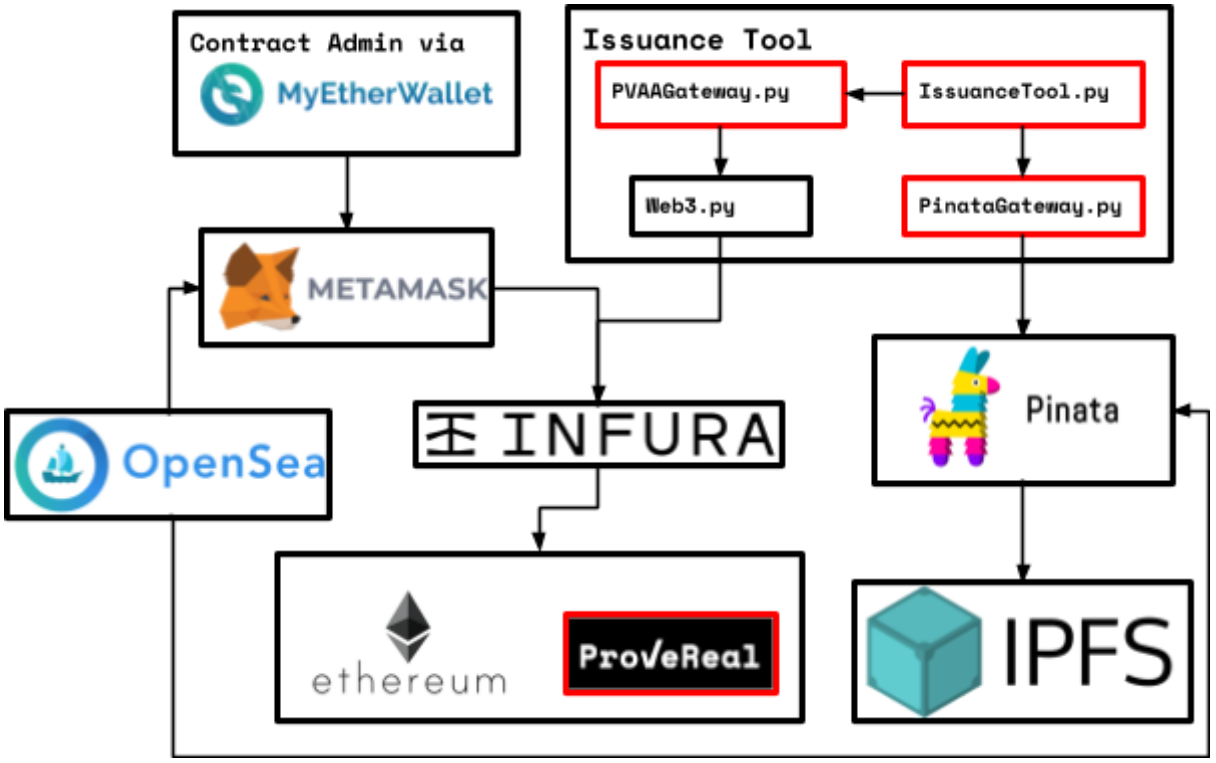
In conjunction to Web3, a wallet implementation is needed to send transactions, in the browser, this is provided using Metamask (Metamask, 2018), a browser plugin that serves as an Ethereum wallet - anybody who wishes to use a website using Web3 requires this plugin.

Web3.py implements the same interfaces as Web3.js, as python Library. This enables the development of blockchain linked applications that can plug into existing ERP systems such as Odoo - which are used highly in supply chain management for large businesses.

Existing Wallets for ERC-721 Tokens

OpenSea and RareBits are web applications that provide wallet interfaces for ERC 721 tokens. Through being compliant with the token standard, tokens produced via our system will viewable through these apps.

Implementation



The implemented code consists of 2 main systems developed for this project : the Python based terminal issuance tool and an ERC721 based smart contract.

The Issuance tool uses Pinata, an IPFS pinning service, to upload data to IPFS in a way such that it will persist once being uploaded to the network. The issuance tool structures metadata to be compliant with ERC-721 metadata standards, this ensures complete compatiblity with ERC-721 Wallets such as OpenSea and Coinbase Wallet, enabling tokens to viewed easily by end users.

The smart contract implements brand and metadata registries to only authorise the issuance of tokens from approved parties, who registered their metadata before use. The system also allows the contract owner to take fees.

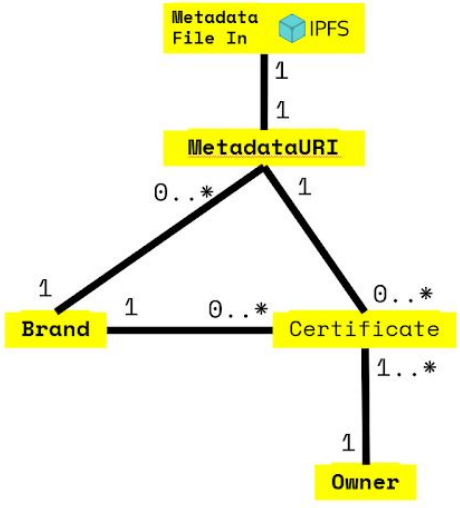
Functionality

The solution implements the the following:

- Maintain a fully ERC-721 compliant non fungible token registry
    - Having a metadata attribute to link to an IPFS URL
    - Having an issuer field, to state the brand that made a given NFT
    - Token holders can view their tokens via OpenSea
- Allow only authorised addresses to issue new NFTs
    - Addresses requesting to make a certificate must be registered brands
    - Brands must register a metadata URI before using it
- Only execute method calls if correct fee has been given
- Maintain a fee schedule for the issuance method
- Allows the contract owner to change fees
- Allow the contract owner to collect fees charged
- Provides a portal for brands to create metadata
- Provides a portal for brands to manually issue certificate

Smart Contract

The smart contract implemented, ProveRealCore, builds a brand registry and metadata registry on top of OpenZeppelin's ERC721Full and ERC721Burnable contracts, which it inherits. ERC721Burnable exposes a public burn (destroy/delete token) feature, that a token holder (for example customer of a shoe brand) can use in a situation where they received the wrong token due to human error, or need to return the item etc. Consumer interfaces for this method were not developed, however, it was tested sucessfully (see appendix A). The OpenZeppelin ownable contract was also inherited, to aid with the setting of permissions.

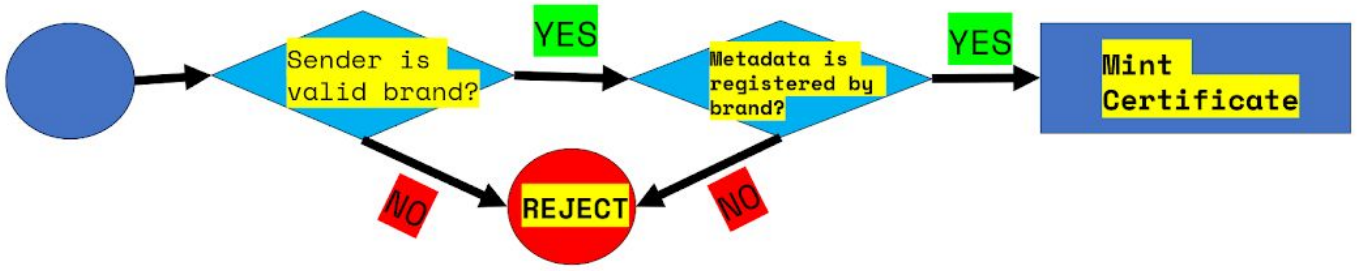


The original intent of the smart contract was to have an additional brand attribute for tokens alongside URIs, however, as this would not be compliant with the ERC-721 metadata standards being used by popular wallets, a different means of representing this relationship was used. The metadataURI is mapped to the brand it belongs to, meaning a way to prove it came from the correct brand can be developed later into a consumer user interface.

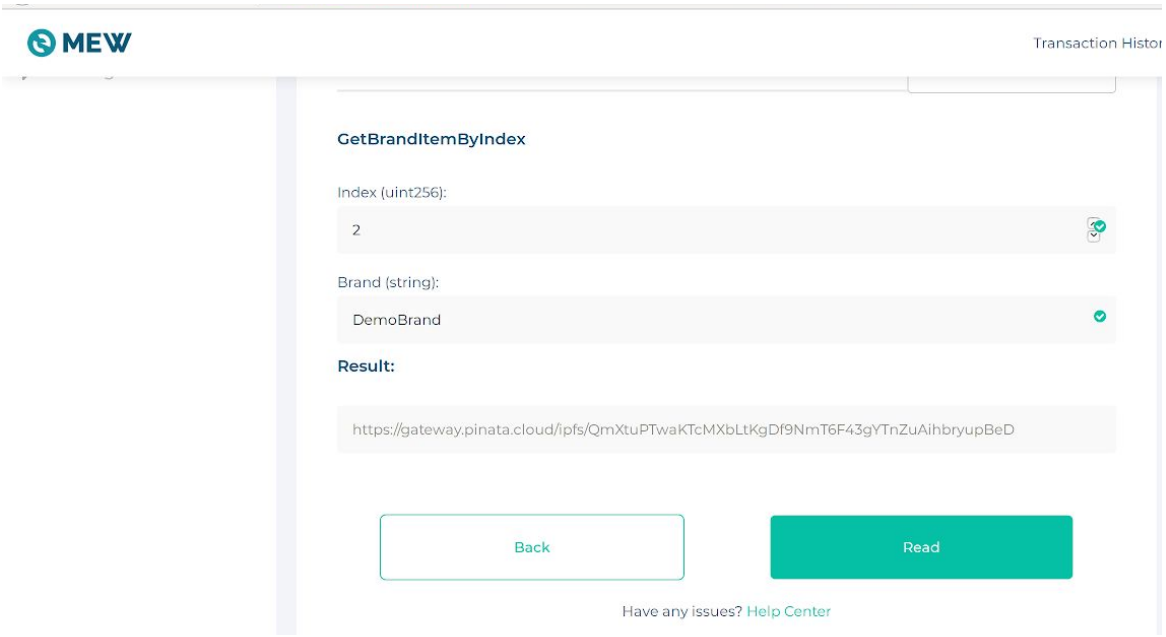
The assumption was made that brand wish to see their provenance, therefore, it was of top concern that they should be able to easily track tokens they issued.

For this reason, a brandToken enumeration had to be made such that a brand could nagivate through all the token IDs of their brand on chain. The brand enumeration system was inspired by that of the _ownedTokens/ _ownedTokensIndex data structures used in OpenZeppelin's ERC721Enumerable (which this contract inherited via ERC721Full). However, instead of using a string => uint[] mapping, I develop an equivalent string => (uint => uint) mapping called _brandMetadataMapping - this was done due to mappings being more gas efficient than arrays in most cases in Ethereum (Gupta 2018) In addition, another string => (string => bool) mapping called _brandHasMetadata was created - to enable gas efficient verification of whether a brand owns a metadata item.

The verification of whether a brand is authorised to use a metadata item is key check in both the registerMetadata and issue methods. Flow chart showing the security measures of used in the issuance of items in used below.



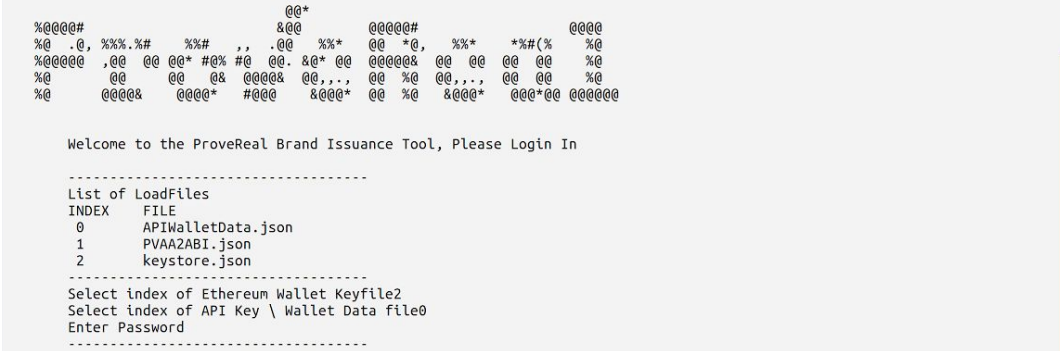
The contract owner can set fees, register brands and withdraw funds from the contract. This can be done via MyEtherWallet, a web based tool for interacting with Ethereum Blockchains. An example of MyEtherWallet accessing the ProveRealCore Contract can be seen below.



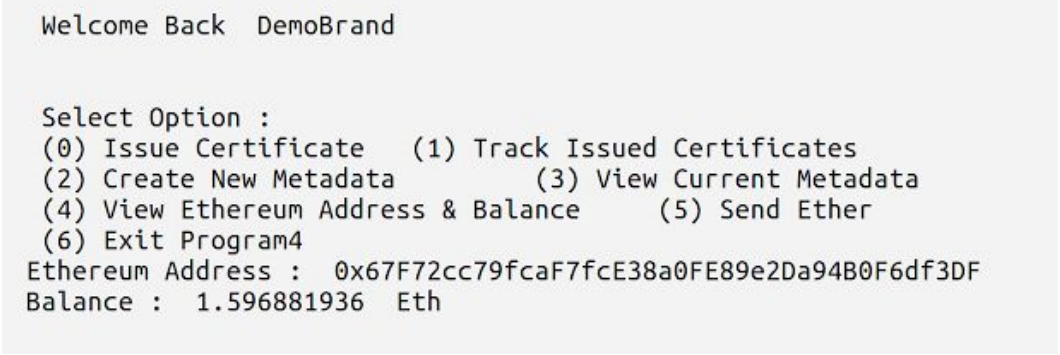
Issuance Tool

The IssuanceTool is a python based wallet enabling registered brands to login to their wallet, and issue metadata and certificates. Additionally, wallet functionality is provided to enable them to receive and send excess Ether.

Below is the menu screen users are greeted with when they wish to log in to the system. They select their wallet keystore, API data files and enter their password to login.

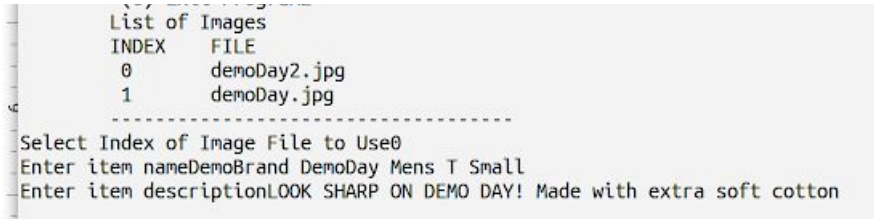


Users can view their balance.

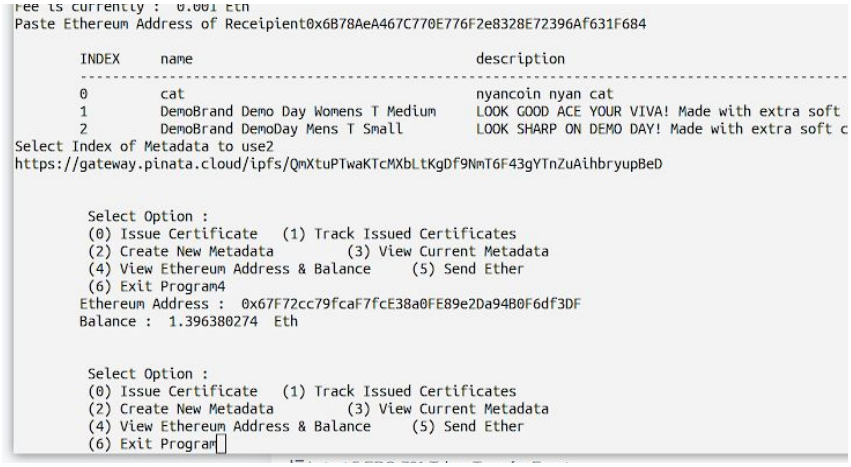


View registered metadata and issued tokens

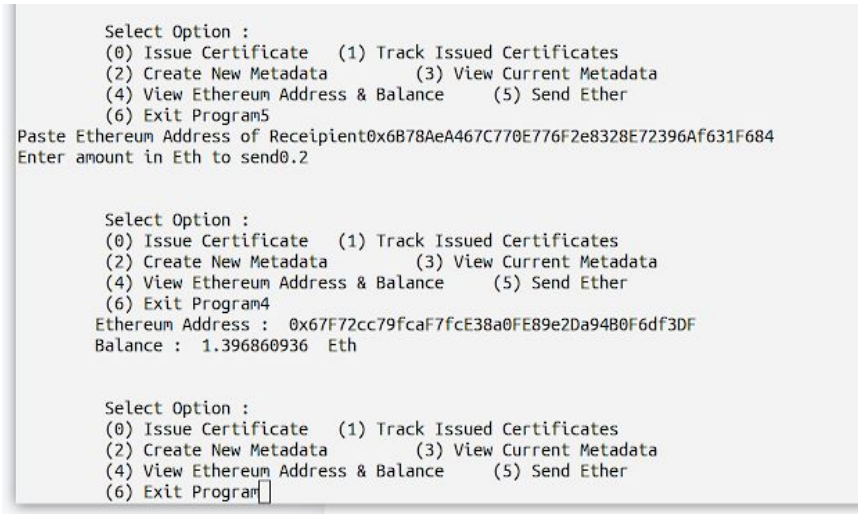
Register metadata



Issue Certificates



Send Ether



IssuanceTool.py utilises PVAAGateway.py and PinataGateway.py to link to the Rinkeby and IPFS respectively. PVAAGateway.py utilises Web3.py to help provide a link to the contract in the Rinkeby network. Due to Web3.py producing a data type called Hex Bytes, which did convert correctly when the int() and str() method were used, method had to be made to reconstruct data obtained by Web3 from the network into data that can be manipulated by python

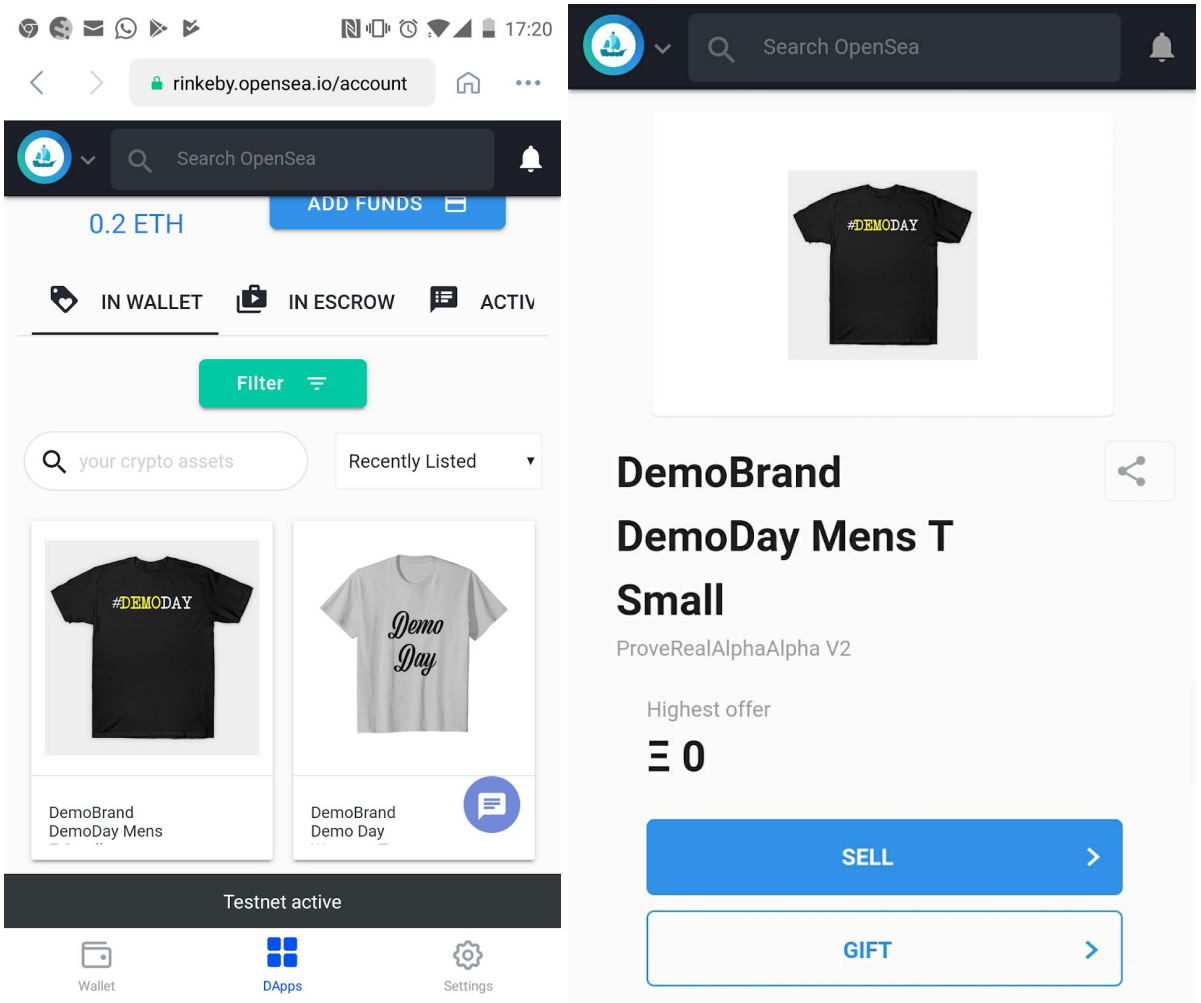
```
# reconstructs a string object from Hex Bytes object produced by Web3.py
def reconstructStr(self,hexBO):
    charList = []
    for i in hexBO:
        if i != 0:
            charList.append(chr(i))
    return ''.join(charList).strip()

# reconstructs an int object from Hex Bytes object produced by Web3.py
def reconstructUInt256(self,hexBO):
    return int.from_bytes(bytes(hexBO),byteorder='big')

# reconstructs an ethereum addres as a string object from Hex Bytes object produced by Web3.py
def reconstructAddress(self,hexBO):
    return '0x'+repr(hexBO)[36:-2]
```

These conversion methods were designed through analysing returned data manually for patterns.

Wallet Functionality via OpenSea

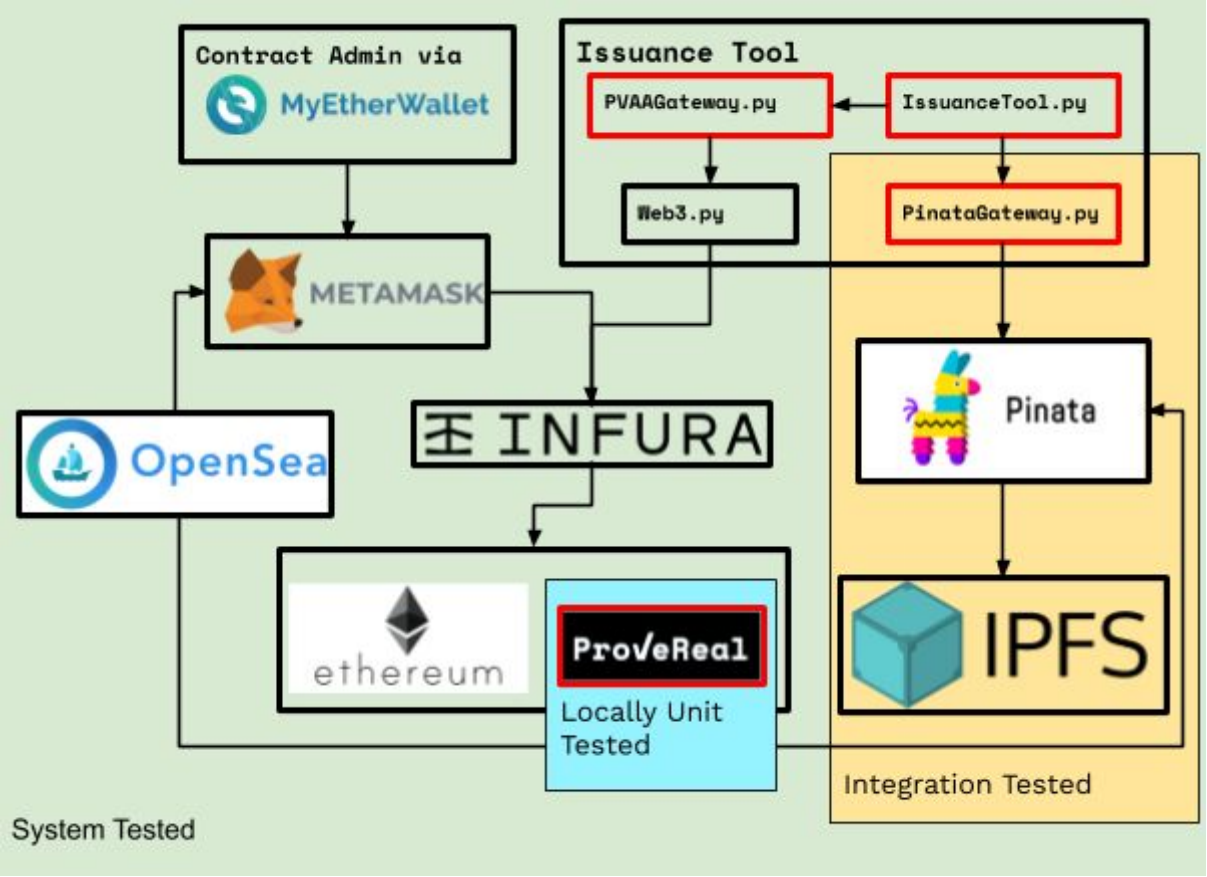
Due to compliance with the ERC-721 Metadata standard, the non-fungible token certificates made using the issuance tool are visible via OpenSea. From here, users can use the functionality of the wallet to transfer the certificate onwards, when selling or gifting the item.

Shown below is OpenSea opened on an Android phone via the Coinbase Wallet app



Testing

Overview



Throughout the development of the platform, the candidate smart contracts were tested via both via truffle based local unit tests, taking place using a local Ganache ethereum test network, and via systems tests, utilising the two versions (initial and final) of the contract deployed to the Rinkeby network.

System testing of the initial contract with IPFS based metadata lead to the discovery of the discovery of the content pinning issue - the key deciding factor in the decision to interface with IPFS via the Pinata API.

All testing took place on a machine running Ubuntu 18.04.1 LTS. An attempt was made to install Web3.py and Truffle on Windows 10, to conduct the testing on a Windows machine. However, the installations failed, this is a problem others have faced.

The system tests took place using the issuance tool, verifying it's user interface functionality works correctly and if it can correctly non fungible token certificates on the Rinkeby testnet which are readable by OpenSea. System testing using PVAAGateway to call the smart contract on the Rinkeby testnet exposed discrepancies between the gas used to execute method calls in a local environment, as opposed to a real distributed network. This is discussed in the section titled Gas Discrepancies.

PinataGateway.py was integration tested with Pinata's own IPFS gateway, as well as the gateway managed by IPFS.io - via a Python unit test producing random input data. Unfortunately, the IPFS.io servers failed to return 30KB files within a 30 second timeout period. However, the 150 byte JSON file dataset, that was tested alongside the 30KB dataset, was successfully returned by IPFS.io. Luckily, due to the fact the issuance tool sets image URLs to point to the gateway managed by Pinata (which return data correctly), this issue does not effect our system. However, this issue does create an element of centrality in the storage of metadata - something the project aimed to avoid, via the use IPFS.

Apart from this issue, the tests proved the codebase successfully carries out its aims - with 46 out of 47 tests being successful.

Test Suite	**Passing**	**Failing**	**Total**
Truffle Unit	26	0	26
Pinata / IPFS Integration	6	1	7
System	14	0	14
Total	**46**	**1**	**47**

Smart Contract Unit Tests

Truffle provides two means of testing contracts within its framework. Via Solidity, and via JavaScript. Solidity based testing takes place entirely on the Ethereum Virtual Machine, to serve as the best means of determining how the tested contract interacts with other smart contracts. JavaScript based testing is used to test how transactions submitted via a wallet client are handled (Consensys 2019). Therefore, JS based unit testing was chosen as it best reflects the use of the smart contract.

When ganache is running it exposes 10 Ethereum accounts that can be accessed via the web3 interface. These accounts are accessed by Truffle to conduct the unit tests. The testing framework is Mocha.js. Before each test there is a setup script which creates a new proveRealCore contract where accounts[0] is the owner. After each test, this contract is killed via the kill() method. Therefore, all tests occur independently of each other as fresh contracts are used each test. This test can be run by calling `truffle test` in the truffle projects root directory. Further detail regarding the test, as well supporting evidence can be found in Appendices A & B.

Passing	Failing	Total
26	0	26

Pinata & IPFS gateway Integration Tests

This test suite aims to prove if the PinataGateway class uploads the correct data to IPFS, and if it retrievable via both Pinata's IPFS gateway, and that of IPFS.io. IPFS.io's gateways are independent of Pinata's and therefore serve as good indication of whether the content is obtainable throughout the network.

The PinataGateway class has no method of determining if the data it is currently handling has been previously uploaded to IPFS - if this is not taken into consideration, regression could be unaccounted for. An example of this could be a request for a given content identifier returning the correct data, not because PinataGateway works correctly, but due to the data already being in the network. Accordingly, data is randomly generated for each test to serve as a precaution.

The suite consists of seven tests written using Python's built in unittest module - which can be tested by running Python3 PinataGateway_test.py in the IssuanceTool directory of the provided source files. The external library "requests" was used to communicate with the IPFS gateways for confirmation. No test fixtures were used. Please see Appendix C and Appendix D for in depth guide to the tests and accompanying results.

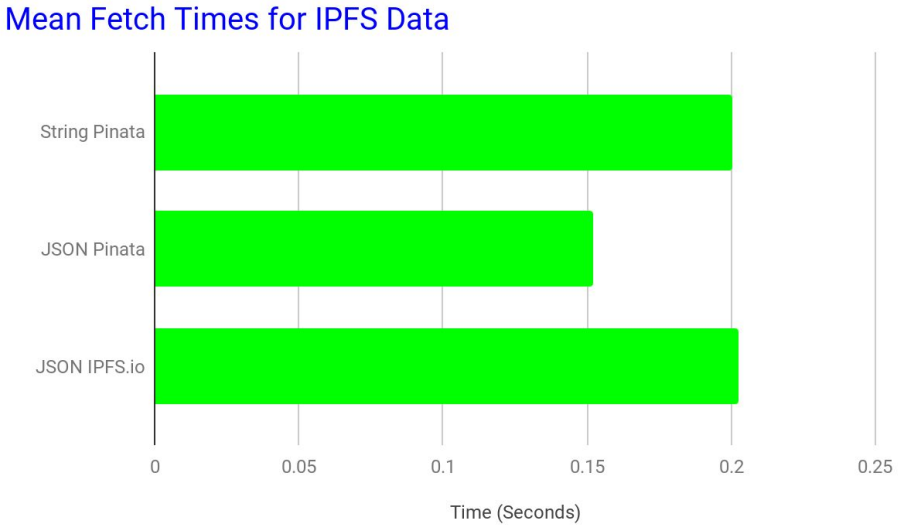
Passing	Failing	Total
6	1	7

Discussion

A key point of interest is that whilst IPFS.io took comparable, but marginally longer, times to return JSON data when compared to Pinata - it completely failed to fetch 30KB files within a 30 second period. One thing to bear in mind is that even though this test failed, it does not have any catastrophic consequences, as the image_url points to Pinata's gateway and not that of IPFS.io.

IPFS.io took roughly 33% longer than Pinata to fetch the same JSON data, with fetch times of 152ms and 202ms, respectively. Obtaining 30KB files from Pinata was slower than obtaining JSON, taking 201ms.

FileType Gateway	Mean Fetch Time (ms)
String Pinata	201
JSON Pinata	152
JSON IPFS.io	202



When data is pinned within Pinata, pinata's servers can return the data immediately - as they do not need to query other IPFS nodes to find the content. Therefore, lookup of the content from in other nodes can be discounted as a factor. To one significant figure, it can be said an additional 50ms was required to load addition 30000 bytes of data, when comparing the JSON dataset to the random strings. 30Kbytes = 240Kbits → 240*(1000/50) = 4.8Mbit/s. Due to this deduction it is reasonable to say the longer wait time for the larger file was due to bandwidth.

Conversely, the discrepancy between JSON retrieval times for both the Pinata IPFS gateway and IPFS.io gateway can be explained by the fact that IPFS.io's servers have to query the rest of the network to find the data pinned by Pinata, and then obtain it. This can explain the longer retrieval times from IPFS.io.(Benet J 2014)
The failure of the IPFS.io gateway to deliver can be put on two things, either : IPFS.io's gateway cannot correctly return the file even though it possesses it; or the IPFS.io gateway cannot obtain the file. As IPFS.io's gateway is well used and battle tested, (IPFS.io 2017), the former is unlikely. This raises the question, why does bigger content fail to propagate throughout the network, when the smaller json files were downloaded highly quickly? This may be due to a feature in the functionality of BitSwap strategy taken by Pinata, where they are less likely to share data the larger it is, and the higher the receiving node is in "debt" (Benet J 2014). This information is not immediately obvious, and research into data analysis of block propagation in the IPFS network would be an interesting topic of research.

The failure of IPFS.io to return the input strings, presents a point of centrality, as it demonstrates image data (which would be of roughly the same size) can only be reliably fetched from Pinata. In theory, more popular image (from extremely coveted trainers, perhaps) on the ProveReal platform would be better seeded in the network, as more nodes would hold that file, due to more clients requesting it (benet, J 2014).

System Tests

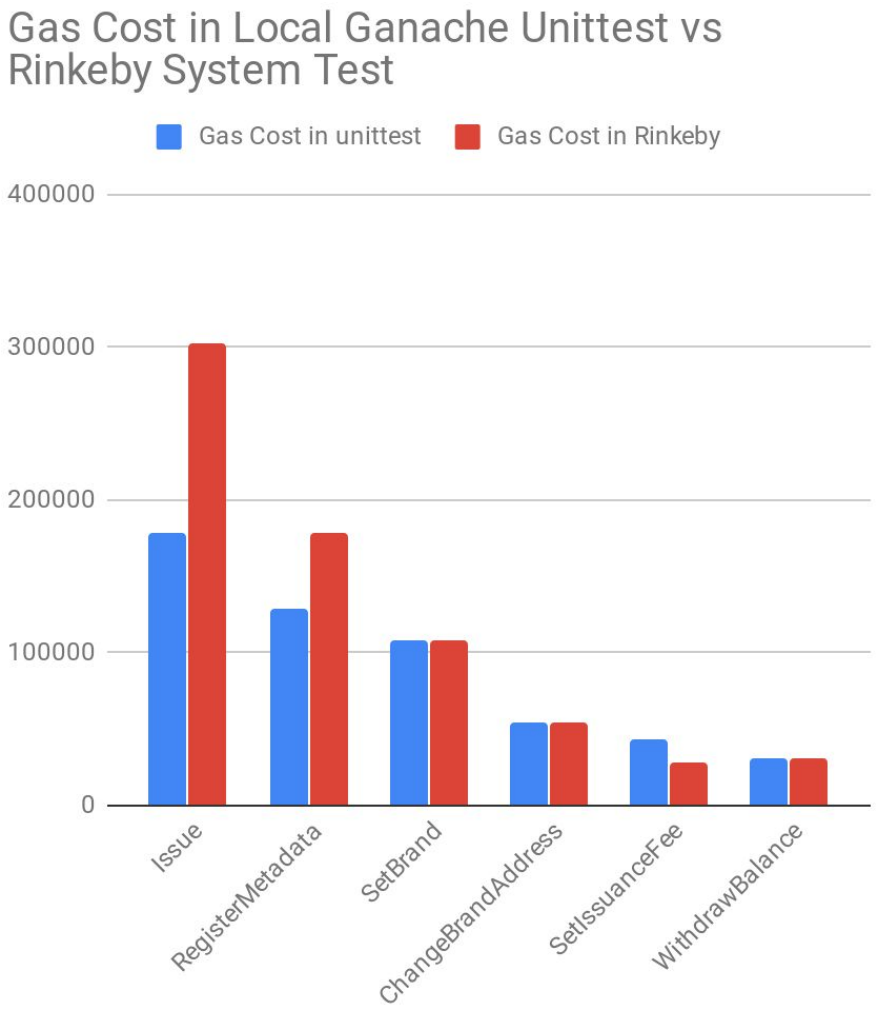
For the system tests, the following actions were taken to setup the system. A keystore file was created using MyEtherWallet's (MEW) (myetherwallet.com) wallet creation tool, the password to this wallet was set to "TestPass01234TestPass". This wallet was named keystore.json and placed in the IssuanceTool directory. The address corresponding to this address was given the brandname "DemoBrand" - this was done via MEW's online contract gateway. The issuance fee was set to 0.001 Eth. The DemoBrand wallet was then logged into the Issuance tool to register two metadata items and three tokens.

Tests were conducted via the IssuanceTool.py program and verified via Pinata, MyEtherWallet, Etherscan, Coinbase Wallet and OpenSea (Appendix E).

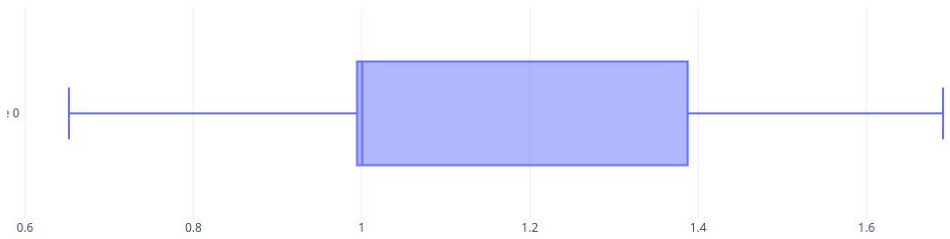
Passing	Failing	Total
14	0	14

Gas Cost Discrepancies

Analysing Ethereum transactions produced during the setup and following the system test, which occurred on the Rinkeby network, lead to the discovery of a gap between the gas cost metered in the local Ganache blockchain used for unittesting and the distributed network the final contract was deployed to. The analysis covered all contract functions which changed the state of the network (see Appendix F).



To determine if the significance of the discrepancy warranted further investigation, the gas costs from the two networks were compared for each method, in order to form a ratio - Rinkeby/Ganache gas cost (see Appendix F). The quartile value were then calculated (see Appendix G) and accordingly used to form a box plot. For the majority of cases in our dataset, the variation is not significant, with Q1 and Q2 having a difference of 0.006 and half of the six methods having a variation of less than 1%. However, Issue, RegisterMetadata and SetIssuance fee, three key methods had variations in excess of 30%.



Gas costs are paid upfront when a transaction is submitted, at a cost of gasLimit*gasPrice. The gas limit is the highest the sender of a transaction is willing to pay to have it executed. The gas cost is a variable cost of Ether per unit gas that one must to have a transaction mined. As we only concerned with units of gas and not cost in terms of Ether, the gas cost is irrelevent. The Ethereum Yellow Paper (Wood 2019) states that the intrinsic gas (g_0) of a transaction(the minimum gas limit required for the transaction begin execution) is equal to:

$$nG_{txdatanonzero} + zG_{txdatazero} + G_{transaction}$$

Where n and z are the number of the non-zero and zero bytes, respectively, within a given transaction submitted to the blockchain. $G_{txdatanonzero}$ is the fee of gas per byte of non zero data, and $G_{txdatazero}$ is the fee for every zero byte. $G_{transaction}$ is the minimum gas cost for any transaction and is the only fee paid when Eth is transferred between accounts. With the curent value of these constants factored in, the gas limit per transaction equates to $68n + 4z + 21000$.

Using this formula on our example issue() transaction on the Rinkeby Network (0xa19bdb3d2ca39f5b9309980d6ebea377c7f5b4e8295ff7bb53d66e9e09c8bb69) [enter this into a Rinkeby blockchain explorer to find the specific transaction], we can deduce this gas limit by finding the number of zero bytes are in the input data, combined with the number of non-zero bytes. This can be done by taking the hexadecimal input of the transaction, as found on Etherscan, and analysing it using Regular Expressions. The number of zero bytes can be found by finding the number of matches for the expression /00/g, let this equal z. Let n be : half the length of the input string minus z.

Doing this calculation on our example input, we get : z = 90 and n = 106. With this we get $g_0 = 106*68 + 90*4 + 21000 = 28568$. This is much lower than the 302,340 gas actually used for the transaction, and in most cases the minimum bound calculated via this formula is too low to serve a means of calculating gas costs.

The intrinsic gas limit serves as an investment one must make to submit a transaction, preventing spam transactions. Much like how Capchas are a proof of work. However, the inclusion of data as a factor is a good indicator of the relative cost of a transaction

given its input data. In fact, from the Ethereum Yellow Paper Fee Schedule (Wood 2019 Appendix G) we can see that in addition to $G_{txdatanonzero}$ and $G_{txdatazero}$, 2 key gas determining constants depend on the size of the input data. These are G_{memory} and $G_{sha3word}$. These constants decide the gas costs of storing new data on the Ethereum Network and hashing data. Hashing occurs a high number of of times in the issue and registerMetadata functions as they make heavy use of mappings - both to store data and verify the existance of other data.

From this we can gather that input data is the key factor driving this difference in gas cost. For example, "https://gateway.pinata.cloud/ipfs/QmXtuPTwaKTcMXbLtKgDf9NmT6F43gYTnZuAihbryupBeD" inputted for our actual system test issuance, which is considerable larger than "metadataURI000", which was used in the local unit test.

Conversely, the size of the data input for changeBrandAddress (2 addresses in each case) and setBrand ("Test01234" vs "DemoBrand") stayed roughly constant. Their diferences in price can be explained by comparing the number of zero and non-zero bytes. WithdrawBalance has no arguments and thus there are no factors which can effect its gas cost.

In summary, the discrepancies found were due to the data being used for these transactions differring from that used in the testing. This can be rectified by building contract unit tests to test with more realistic data.

Further Work

The contract could be optimised further by reducing the data processing requirements of the code running on the network. One way this could be done is by modifying 2D mappings of type a => (a=>a) to be (a=>a)=>a. This would significantly reducing the data hashing required, leading to a reduction in the cost of gas.

Conclusion

This project presents practical use for blockchain, in the prevention of the counterfeiting of Luxury items. The smart contract and the accompanying issuance tool and their underlying components were throughly tested and proven to work to a high standard - producing a solution compliant with existing protocols.

References

Helmore, E. (2018). *'Brands are the new bands' – Hypefest and how streetwear got its own festival*. [online] the Guardian. Available at: https://www.theguardian.com/fashion/2018/oct/09/brands-are-the-new-bands-hypefest-and-how-streetwear-got-its-own-festival-new-york

Isichei, I. (2018). *Why Supreme Reigns Supreme*. [online] The Bandwagon. Available at: https://thebandwagon.blog/2018/07/19/why-supreme-reigns-supreme/

The Guardian (2017). *Skate of style ... (Clockwise from top) Supreme Kermit tee; Palace baseball cap; queue at London's Palace store; two Palace enthusiasts; Veterments anorak; Supreme Obama hoodie; Palace Elton tee. Composite: Getty Images/David Levene*. [image] Available at: https://www.theguardian.com/fashion/2017/mar/29/how-streetwear-styled-the-world-from-hip-hop-to-supreme-and-palace

Strijbos, B. (2016). *Global fashion industry statistics - International apparel*. [online] Fashionunited.com. Available at: https://fashionunited.com/global-fashion-industry-statistics

Lea, E. (2018). *Urban Streetwear — The Ignored Multi-Billion Dollar Industry That Millennials are Ruling - [FKD]*. [online] [GenFKD]. Available at: http://www.genfkd.org/urban-streetwear-ignored-multi-billion-dollar-industry-millennials-ruling

The Fashion Law. (2018a). *The Counterfeit Report: The Big Business of Fakes*. [online] Available at: http://www.thefashionlaw.com/home/the-counterfeit-report-the-impact-on-the-fashion-industry

The Fashion Law. (2018b) What is Blockchain and What Can it Do for the Fashion Industry? [online] Available at: http://www.thefashionlaw.com/home/what-is-blockchain-and-what-can-it-do-for-the-fashion-industry

Stanley, J. (2018). *Nike to Close Down 20 Fake Sneaker Websites*. [online] HYPEBEAST. Available at: https://hypebeast.com/2018/5/nike-fake-sneaker-sites

Mantor, C. (2018). Off-White files multi-million dollar lawsuit against Wish.com for selling fakes. [online] FashionNetwork.com. Available at: https://uk.fashionnetwork.com/news/Off-White-files-multi-million-dollar-lawsuit-against-Wish-com-for-selling-fakes,965242.html#.XAldRdJ4rtU

Dejolde, A. (2013). *Genuine or Fake? - How to Spot Fake Fashion Items*. [online] Lifehack. Available at: https://www.lifehack.org/articles/lifestyle/genuine-fake-how-spot-fake-fashion-items.html

We Went To The Fake Sneaker Capital Of China (HBO). (2018). [video] Directed by D. Thomas. Vice News.

Moncler (2016). MONCLER AND THE BATTLE AGAINST COUNTERFEITING [online]. Available at: https://www.monclergroup.com/wp-content/uploads/2016/07/MONCLER-AND-THE-BATTLE-AGAINST-COUNTERFEITING-ENG.pdf

Moncler Expert (2018). QR Code [online]. Available at: http://www.monclerexpert.com/qrcode.html

AsiaRFID (2017). RFID Tags RFID Card RFID Wristbands Supplier. UNIQLO Announced to bring RFID Labels to the 3000 stores within one year. [online] Available at: http://www.asiarfid.com/rfid-news/uniqlo-announced-to-bring-rfid-labels-to-the-3000-stores-within-one-year.html

Bonafi (2018). Whitepaper v4.1 [online] Available at: https://www.bonafi.io/wp-content/uploads/2018/08/Bonafi_Whitepaper_4.1.pdf

Grailed (2018) Protection [online]. Available at: https://www.grailed.com/protection [accessed 6 Dec. 2018]

Stockx (2018) How it Works [online]. Available at: https://stockx.com/how-it-works [accessed 6 Dec. 2018]

Kharpal, A. (2018). Everything you need to know about the blockchain. [online] CNBC. Available at: https://www.cnbc.com/2018/06/18/blockchain-what-is-it-and-how-does-it-work.html

Phys.org. (2018). Visa says over 5 million payments affected by June outage. [online] Available at: https://phys.org/news/2018-06-visa-million-payments-affected-june.html

Colored Coins (2018) Colored Coins [online] Available at: http://coloredcoins.org [accessed 6 Dec. 2018]

Ethereum (2018) Ethereum [online] Available at: http://www.ethereum.org [accessed 6 Dec. 2018]

Ethlend (2018) Ethlend [online] Available at: https://ethlend.io [accessed 6 Dec. 2018]

Wood, G (2019) Ethereum Yellow Paper [online] Available at: https://ethereum.github.io/yellowpaper/paper.pdf [accessed 13/01/19]

MakerDAO (2018) MakerDAO [online] Available at: https://makerdao.com [accessed 6 Dec. 2018]

Open Zeppelin (2018a) Open Zeppelin [online] Available at: https://openzeppelin.org [accessed 6 Dec. 2018]

Cryptokitties (2018) Cryptokitties [online] Available at: https://cryptokitties.co [accessed 6 Dec. 2018]

Token (2018) Token [online] Available at: https://tokencard.io [accessed 6 Dec. 2018]

Etherscan (2018) Etherscan [online] Available at: https://etherscan.io [accessed 1 Nov. 2018]

ETH Gas Station (2018) ETH Gas Station [online] Available at: https://ethgasstation.info [accessed 1 Nov. 2018]

IPFS (2018) IPFS [online] Available at: https://ipfs.io [accessed 6 Dec. 2018]

Ethereum Name Service (2018a) Ethereum Name Service [online] Available at: https://ens.domains [accessed 6 Dec. 2018]

Kaiser, I. (2017). A Decentralised Private Marketplace: DRAFT 0.1. [online] Available at: https://github.com/particl/whitepaper/blob/master/decentralized-private-marketplace-draft-0.1.pdf

Draglet (2018) Why Most New Tokens are Ethereum ICOs. [online] Available at: https://www.draglet.com/why-ethereum-icos/ [accessed 6 Dec. 2018]

Rivlin, B. (2016) Geth, Viper, and Wafr: New Ethereum Developments. [online] Available at: https://www.ethnews.com/geth-viper-and-wafr-new-ethereum-developments

Solidity.readthedocs.io. (2018). Units and Globally Available Variables — Solidity 0.5.1 documentation. [online] Available at: https://solidity.readthedocs.io/en/v0.5.1/units-and-global-variables.html [Accessed 6 Dec. 2018].

Investopedia. (2017). What is ERC-20 and What Does it Mean for Ethereum?. [online] Available at: https://www.investopedia.com/news/what-erc20-and-what-does-it-mean-ethereum/ [Accessed 6 Dec. 2018].

Falkon, S. (2017) The Story of the DAO — Its History and Consequences [online] Available at: https://medium.com/swlh/the-story-of-the-dao-its-history-and-consequences-71e6a8a551ee

Vergauwen, N. (2018) Upgradeable Smart Contracts [online] Available at: https://hackernoon.com/upgradeable-smart-contracts-a7e9aef76fdd

Grincalaitis, M. (2017) The ultimate guide to audit a Smart Contract + Most dangerous attacks in Solidity [online] Available at: https://medium.com/ethereum-developers/how-to-audit-a-smart-contract-most-dangerous-attacks-in-solidity-ae402a7e7868

Zeppelin (2018) Security Audits [online] Available at: https://zeppelin.solutions/security-audits/ [Accessed 6 Dec. 2018].

Breen, A. (2018) How and Why Developing for Ethereum Sucks [online] Available at: https://medium.com/@aidobreen/how-and-why-developing-for-ethereum-sucks-1ff1a9873527

Davies, A. (2018) How to Deploy Smart Contract on Ethereum [online] Available at: https://www.devteam.space/blog/how-to-deploy-smart-contract-on-ethereum/

Wesley, D. (2017) Part 2: Deploying smart contracts in the Browser with Web3JS and vanilla JavaScript. [online] Available at: https://medium.com/@JusDev1988/part-2-deploying-smart-contracts-in-the-browser-with-web3js-and-vanilla-javascript-f85214113fec

GitHub. (2018). Ethereum. [online] Available at: https://github.com/ethereum [Accessed 6 Dec. 2018].

Consensys (2018a) Truffle Overview [online] Available at: https://truffleframework.com/docs/truffle/overview

Status (2018) Embark [online] Available at: https://embark.status.im [Accessed 6 Dec. 2018].

POA Network (2018) POA Network [online] Available at: https://poa.network [Accessed 6 Dec. 2018].

Thundercore (2018) Thundercore [online] Available at: https://www.thundercore.com [Accessed 6 Dec. 2018].

GoChain (2018) If 1000 DApps on Ethereum Need to do 1 Transaction per Second, What Happens? [online] Available at: https://medium.com/gochain/if-1000-dapps-on-ethereum-need-to-do-1-transaction-per-second-what-happens-9e6247e0beca

Gastracker.io (2018) Gastracker.io [online] Available at: https://gastracker.io [Accessed 6 Dec. 2018].

Consensys (2018b) The State of Scaling Ethereum [online] Available at: https://media.consensys.net/the-state-of-scaling-ethereum-b4d095dbafae

Web3js.readthedocs.io (2018) web3.js - Ethereum JavaScript API [online] Available at: https://web3js.readthedocs.io/en/1.0/

Quintal, J. (2017) ETHEREUM PET SHOP -- YOUR FIRST DAPP [online] Available at: https://truffleframework.com/tutorials/pet-shop

Metamask (2018) Metamask [online] Available at: https://metamask.io

Glazer, P. (2018) An Overview of Non Fungible Tokens [online] Available at: https://hackernoon.com/an-overview-of-non-fungible-tokens-5f140c32a70a

Hamilton, D. (2018) The Growing World of Non-Fungible Tokens [online] Available at: https://coincentral.com/non-fungible-tokens/

Open Zeppelin (2018b) ERC721 [online] Available at: https://github.com/OpenZeppelin/openzeppelin-solidity/tree/master/contracts/token/ERC721 [Accessed 6 Dec. 2018].

Simon, G. (2017) Solidity Mappings & Structs Tutorial [online] Available at: https://coursetro.com/posts/code/102/Solidity-Mappings-&-Structs-Tutorial

Virk R (2018) The One Thing Missing from the ERC 721 Standard for Digital Collectibles on the Blockchain [online] Available at: https://hackernoon.com/the-one-thing-missing-from-erc-721-standard-for-digital-collectibles-on-the-blockchain-9ee26e4a918c

Ethereum Name Service (2018b) Subdomain Registrar [online] Available at: https://github.com/ensdomains/subdomain-registrar

Ethereum Name Service (2018c) Interacting with the ENS registry [online] Available at: https://ens.readthedocs.io/en/latest/userguide.html#interacting-with-the-ens-registry

Coin Market Cap (2018) Ethereum [online] Available at: https://coinmarketcap.com/currencies/ethereum/ [Accessed 6 Dec. 2018].

Open Zeppelin (2018c) Ownable [online] Available at: https://openzeppelin.org/api/docs/ownership_Ownable.html

Solidity.readthedocs.io (2018b) Inheritance [online] Available at: https://solidity.readthedocs.io/en/v0.4.24/contracts.html#inheritance [Accessed 6 Dec. 2018].

Solidity.readthedocs.io (2019) Mapping Types [online] Available at: https://solidity.readthedocs.io/en/v0.5.7/types.html#mappings [Accessed 29 Mar. 2019].

Bearne, S. (2017). Meet the teens making thousands from selling online. [online] the Guardian. Available at: https://www.theguardian.com/fashion/2017/oct/23/teens-selling-online-depop-ebay [Accessed 29 Mar. 2019].

Nicholas, T., Russo, M. and Zettler, J. (2018). Token Metadata JSON Schema. [online] Ethereum Improvement Proposals. Available at: http://eips.ethereum.org/EIPS/eip-1047 [Accessed 29 Mar. 2019].

Entriken, T., Shirley, D., Evans, J. and Sachs, N. (2018) Ethereum Improvement Proposals. Available at:

http://eips.ethereum.org/EIPS/eip-721 [Accessed 29 Mar. 2019].

Docs.opensea.io. (2019). 2. Adding metadata. [online] Available at: https://docs.opensea.io/docs/2-adding-metadata [Accessed 29 Mar. 2019].

Docs.rarebits.io (2019) Listing your Assets [online] Available at: https://docs.rarebits.io/v1.0/docs/listing-your-assets [Accessed 29 Mar. 2019].

Business Wire. (2019). Ralph Lauren Reports Third Quarter Fiscal 2019 Results. [online] Businesswire.com. Available at: https://www.businesswire.com/news/home/20190205005234/en/Ralph-Lauren-Reports-Quarter-Fiscal-2019-Results [Accessed 1 Apr. 2019].

Stone Island. (2019) Stone Island UK | Authenticity [online] Available at: https://www.stoneisland.co.uk/pages/authenticity [Accessed 1 Apr. 2019].

Alvarez E, (2017) [online] How RFID tags became trendy [online] Available at : https://www.engadget.com/2017/08/22/rfid-tags-in-fashion/?guccounter=1&guce_referrer_us=aHR0cHM6Ly93d3cuZ29vZ2xlLmNvLnVrLw&guce_referrer_cs=Ky4LAA45ncq1gMVftptXrg [Accessed 1 Apr. 2019].

Certilogo (2019a) [online] Certilogo Available at : https://www.certilogo.com [Accessed 1 Apr. 2019].

Certilogo (2019b) [online] FAQ Available at :https://www.certilogo.com/faq [Accessed 1 Apr. 2019].

Reindl, J. (2018). Fast-growing Detroit startup StockX sniffs out fake sneakers. [online] Eu.freep.com. Available at: https://eu.freep.com/story/money/business/2018/07/09/detroit-stockx-sniffs-out-fake-sneakers/731070002/ [Accessed 1 Apr. 2019].

Mitchell K (2018) Photo of StockX Authenticator [online] https://www.gannett-cdn.com/media/2018/07/06/DetroitFreeP/DetroitFreePress/636664682429480012-stockX-070318-kpm-154.jpg?width=1080&quality=50 [Accessed 1 Apr. 2019].

Schmitt, L. (2018). Centrifuge Tokenized Invoices as Collateral For Dharma Loans. [online] Medium. Available at: https://medium.com/centrifuge/centrifuge-tokenized-invoices-as-collateral-for-dharma-loans-b214a23bc66 [Accessed 1 Apr. 2019].

Fernandez, C. (2018). Luxury as the Next Stock Market of Things: Josh Luber. [online] The Business of Fashion. Available at: https://www.businessoffashion.com/articles/video/stockx-is-luxury-next-on-the-stock-market-of-things-josh-luber [Accessed 1 Apr. 2019].

Grailed (2019) Buyer and Seller Protection [online] Grailed Available at: https://www.grailed.com/protection#our-moderators [Accessed 1 Apr. 2019].

Amazon (2019) Amazon Project Zero [online] Amazon Available at : https://brandservices.amazon.com/projectzero [Accessed 1 Apr. 2019].

Provenance(2019) A Platform for Business [online] Provenance Available at : https://www.provenance.org/business/platform [Accessed 1 Apr. 2019].

Allison, I (2019) Louis Vuitton Owner LVMH Is Launching a Blockchain to Track Luxury [online] Available at : Goodshttps://www.coindesk.com/louis-vuitton-owner-lvmh-is-launching-a-blockchain-to-track-luxury-goods [Accessed 1 Apr. 2019].

Benet, J (2014) IPFS - Content Addressed, Versioned, P2P File System(DRAFT 3) [online] Available at: https://ipfs.io/ipfs/QmV9tSDx9UiPeWExXEeH6aoDvmihvx6jD5eLb4jbTaKGps [Accessed 5 Apr 2019]

(IPFS.io, 2017) Uncensorable Wikipedia on IPFS [online] Available at: https://blog.ipfs.io/24-uncensorable-wikipedia/ [Accessed 20 Apr 2019]

Consensys (2019)https://truffleframework.com/docs/truffle/testing/testing-your-contracts [Accessed 1 Mar 2019]

Nakamoto (2009) Bitcoin A peer to peer cash system [online] Available at : https://bitcoin.org/bitcoin.pdf

Ethereum (2019) Ethereum Whitepaper [online] Available at : https://github.com/ethereum/wiki/wiki/White-Paper

Douceur J.R. (2002) The Sybil Attack. In: Druschel P., Kaashoek F., Rowstron A. (eds) Peer-to-Peer Systems. IPTPS 2002. Lecture Notes in Computer Science, vol 2429. Springer, Berlin, Heidelberg

Block'tivity (2019) *The real value of Blockchains [online] Available at:* https://blocktivity.info/

Zheng X (2017) Phishing with Unicode Domains [online] Available at: https://www.xudongz.com/blog/2017/idn-phishing/

Ethereum Name Service (2019) Frequently Asked Questions [online] Available at: https://docs.ens.domains/frequently-asked-questions

Gupta, M (2018) Solidity gas optimisation tips [online] Available at : https://mudit.blog/solidity-gas-optimization-tips/

AppendixA: Truffle Test Table

Test Description	Test Code	Result
When initialised, the contract's fee value is zero. This test calls setIssuanceFee using the contract owner address, to set the fee to 1 Eth. Test successful if the issuanceFee getter 1 Eth.	`it("setIssuanceFee correctly sets fees when requested by contract owner", async()=>{ let t1 = await proveRealCoreInstance.setIssuanceFee(new BigNumber(web3.utils.toWei("1","ether"))); console.log({'setIssuanceFee gas used' : t1.receipt.gasUsed}); let issuanceFee = await proveRealCoreInstance.issuanceFee(); assert.equal(issuanceFee.toString(),web3.utils.toWei("1","ether").toString() ,"issuanceFee updated correctly"); });`	
Attempts to call setIssuanceFee using a non owner account TruffleAssert.reverts declares the test a success if I catches an exception	`it("setIssuanceFee prevents non contract owner from changing issuance fee", async()=>{ await truffleAssert.reverts(proveRealCoreInstance.setIssuanceFee(new BigNumber(web3.utils.toWei("1","ether")),{from:accounts[1]})); });`	
Sets up a brand "Test01234" assigned to accounts[1] by calling setBrand using the owner address. Test is successful if a subsequent setBrand call assigning "Test01234" to accounts[2] reverts.	`it("setBrand prevents contract owner from overwritting a registered brand", async()=>{ let t1 = await proveRealCoreInstance.setBrand(accounts[1],"Test01234"); await truffleAssert.reverts(proveRealCoreInstance.setBrand(accounts[2],"Test01234" )); });`	
Sets up a brand "Test01234" assigned to accounts[1] by calling setBrand using the owner address. Test is successful if a subsequent setBrand call by accounts[2] assigning "Test01234" to itself reverts.	`it("setBrand prevents non contract owners from overwritting a registered brand", async()=>{ let t1 = await proveRealCoreInstance.setBrand(accounts[1],"Test01234"); await truffleAssert.reverts(proveRealCoreInstance.setBrand(accounts[2],"Test01234" ,{from:accounts[2]})); })`	
Sets up a brand "Test01234" assigned to accounts[1] by calling setBrand using the owner address.	`it("setBrand works correctly when called by contract owner", async()=>{ let t1 = await`	

Test successful if addressToBrand(accounts[1]) returns “Test01234” and brandToAddress	proveRealCoreInstance.setBrand(accounts[1],"Test01234"); console.log({'setBrand gas used' : t1.receipt.gasUsed}); let brandName = await proveRealCoreInstance.addressToBrand(accounts[1]); assert.equal(brandName,"Test01234","address points to brand"); let address = await proveRealCoreInstance.brandToAddress("Test01234") assert.equal(accounts[1],address,"brand points to correct address") });	
Test is successful if a setBrand call by accounts[1] assigning “Test01234” to itself reverts.	it("setBrand prevents non contract owner from setting up a brand", async()=>{ await truffleAssert.reverts(proveRealCoreInstance.setBrand(accounts[1],"Test01234",{from:accounts[1]})); });	
Sets up a brand “Test01234” assigned to accounts[1] by calling setBrand using the owner address. Test Succesful if subsequent call to setBrand, also giving accounts[1] control of “Test56789”	it("setBrand prevents the registration of a brand to an address already owning a brand", async()=>{ let t1 = await proveRealCoreInstance.setBrand(accounts[1],"Test01234"); await truffleAssert.reverts(proveRealCoreInstance.setBrand(accounts[1],"Test56789")); })	
Sets up a brand “Test01234” assigned to accounts[1] by calling setBrand using the owner address. ChangeBrandAddress is then called by the contract owner address - to assign accounts[2] the brand held by accounts[1] Test successful if addressToBrand(accounts[2]) returns “Test01234” and addressToBrand(accounts[1]) returns a blank string. Indicating accounts[2] now represents the brand and accounts[1] is linked to nothing	it("changeBrandAddress works correctly when called by contract owner", async()=>{ let t1 = await proveRealCoreInstance.setBrand(accounts[1],"Test01234"); let t3 = await proveRealCoreInstance.changeBrandAddress(accounts[1],accounts[2]); console.log({'changeBrandAddress gas used' : t3.receipt.gasUsed}); let brandName2 = await proveRealCoreInstance.addressToBrand(accounts[2]); assert.equal(brandName2,"Test01234","changeBrandAddress works correctly"); let brandName3 = await proveRealCoreInstance.addressToBrand(accounts[1]); assert.equal(brandName3,"","Former brand addresses do not point to the brand name"); });	
Sets up a brand “Test01234” assigned to accounts[1] by calling setBrand using the owner address. ChangeBrandAddress is then called by accounts[1] - to assign accounts[2] the brand it holds. Test successful if addressToBrand(accounts[2]) returns “Test01234” and addressToBrand(accounts[1]) returns a blank string. Indicating accounts[2] now represents the brand and accounts[1] is linked to nothing	it("changeBrandAddress works correctly when called by brand owner", async()=>{ let t1 = await proveRealCoreInstance.setBrand(accounts[1],"Test01234"); let t3 = await proveRealCoreInstance.changeBrandAddress(accounts[1],accounts[2],{from:accounts[1]}); let brandName2 = await proveRealCoreInstance.addressToBrand(accounts[2]); assert.equal(brandName2,"Test01234","changeBrandAddress works correctly"); let brandName3 = await proveRealCoreInstance.addressToBrand(accounts[1]); assert.equal(brandName3,"","Former brand addresses do not point to the brand name"); });	
Sets up a brand “Test01234” assigned to accounts[1] by calling setBrand using the owner address. Test successful if a subsequent call to	it("changeBrandAddress prevents non contract owner parties to change brand addresses for brands they do not own",async()=>{ let t1 = await proveRealCoreInstance.setBrand(accounts[1],"Test01234");	

changeBrand by accounts[2] attempting to transfer ownership of account[1]'s brand to itself fails.	`await truffleAssert.reverts(proveRealCoreInstance.changeBrandAddress(accounts[1],accounts[2],{from:accounts[2]}));` `});`	
Sets up a brand "Test01234" assigned to accounts[1] by calling setBrand using the owner address. Sets up a brand "Test56789" assigned to accounts[2] by calling setBrand using the owner address. Test successful if a changeBrand call from the contract owner to assign accounts[1]'s brand to account[2] fails	`it("changeBrandAddress prevents the changing of a brand address to an address already owning another brand", async()=>{` `let t1 = await proveRealCoreInstance.setBrand(accounts[1],"Test01234");` `let t2 = await proveRealCoreInstance.setBrand(accounts[2],"Test56789");` `await truffleAssert.reverts(proveRealCoreInstance.changeBrandAddress(accounts[1],accounts[2]));` `});`	
Sets up a brand "Test01234" assigned to accounts[1] by calling setBrand using the owner address. Accounts[1] calls the registerMetadata method to register "metadataURI000" Test successful if getBrandItemByIndex and getMetadata index return complementary values.	`it("registerMetadata adds one new entry correctly", async()=>{` `let t1 = await proveRealCoreInstance.setBrand(accounts[1],"Test01234");` `let t2 = await proveRealCoreInstance.registerMetadata("metadataURI000",{from:accounts[1]});` `console.log({'registerMetadata gas used ' : t2.receipt.gasUsed});` `let metadata = await proveRealCoreInstance.getBrandItemByIndex(0,"Test01234");` `assert(metadata,"metadataURI000","Metadata correctly obtained from index in brand metadata mapping");` `let metadataID = await proveRealCoreInstance.getMetadataIndex("metadataURI000");` `assert(metadataID,0,"Correct metadataID returned");` `});`	
Sets up a brand "Test01234" assigned to accounts[1] by calling setBrand using the owner address. Accounts[1] calls the registerMetadata method to register "metadataURI000" Test successful if a subsequent call to registerMetadata by accounts[1], to register "metadataURI000" again reverts .	`it("registerMetadata prevents brands from double entering metadata they already registered", async()=> {` `let t1 = await proveRealCoreInstance.setBrand(accounts[1],"Test01234");` `let t2 = await proveRealCoreInstance.registerMetadata("metadataURI000",{from:accounts[1]});` `await truffleAssert.reverts(proveRealCoreInstance.registerMetadata("metadataURI000",{from:accounts[1]}),"");` `});`	
Sets up a brand "Test01234" assigned to accounts[1] by calling setBrand using the owner address. Sets up a brand "Test56789" assigned to accounts[2] by calling setBrand using the owner address. Accounts[1] calls the registerMetadata method to register "metadataURI000" Test successful if a subsequent call to registerMetadata by accounts[2], to register "metadataURI000" reverts .	`it("registerMetadata prevents brands from registering metadata registered by another brand", async()=> {` `let t1 = await proveRealCoreInstance.setBrand(accounts[1],"Test01234");` `let t2 = await proveRealCoreInstance.setBrand(accounts[2],"Test56789");` `let t3 = await proveRealCoreInstance.registerMetadata("metadataURI000",{from:accounts[1]});` `await truffleAssert.reverts(proveRealCoreInstance.registerMetadata("metadataURI000",{from:accounts[2]}));` `});`	
Sets up a brand "Test01234" assigned to accounts[1] by calling setBrand using the owner address. Accounts[1] calls the registerMetadata	`it("getBrandMetadataCount returns the expected value", async()=> {` `await proveRealCoreInstance.setBrand(accounts[1],"Test01234");` `await proveRealCoreInstance.registerMetadata("metadataURI000",{from:accounts[1]});`	

method 3 times to register 3 items of metadata. Test sucessful if getBrandMetadataCount indicates that “Test01234” has 3 items of metadata	`await proveRealCoreInstance.registerMetadata("metadataURI001",{from:accounts[1]});` `await proveRealCoreInstance.registerMetadata("metadataURI002",{from:accounts[1]});` `let metadataCount = await proveRealCoreInstance.getBrandMetadataCount("Test01234");` `assert.equal(metadataCount,3,"correct value returned")` `});`	
Sets up a brand “Test01234” assigned to accounts[1] by calling setBrand using the owner address. Accounts[1] calls the registerMetadata method 3 times to register 3 items of metadata. Test sucessful if isMetadataRegistered returns true for every string accounts[1] registered.	`it("isMetadataRegistered returns correct", async()=> {` `await proveRealCoreInstance.setBrand(accounts[1],"Test01234");` `await proveRealCoreInstance.registerMetadata("metadataURI000",{from:accounts[1]});` `await proveRealCoreInstance.registerMetadata("metadataURI001",{from:accounts[1]});` `await proveRealCoreInstance.registerMetadata("metadataURI002",{from:accounts[1]});` `let md0 = await proveRealCoreInstance.isMetadataRegistered("metadataURI000");` `let md1 = await proveRealCoreInstance.isMetadataRegistered("metadataURI001");` `let md2 = await proveRealCoreInstance.isMetadataRegistered("metadataURI002");` `assert(md0,"");` `assert(md1,"");` `assert(md2,"");` `});`	
Sets up a brand “Test01234” assigned to accounts[1] by calling setBrand using the owner address. Sets up a brand “Test56789” assigned to accounts[2] by calling setBrand using the owner address. Each brand calls registerMetadata to register 3 items of metadata each. Test successful if each incremental call to getBrandItembyIndex returns the items in the order they were registered in	`it("getBrandItemByIndex works correctly", async()=> {` `await proveRealCoreInstance.setBrand(accounts[1],"Test01234");` `await proveRealCoreInstance.setBrand(accounts[2],"Test56789");` `await proveRealCoreInstance.registerMetadata("metadataURI000",{from:accounts[1]});` `await proveRealCoreInstance.registerMetadata("metadataURI001",{from:accounts[1]});` `await proveRealCoreInstance.registerMetadata("metadataURI002",{from:accounts[1]});` `await proveRealCoreInstance.registerMetadata("metadataURI010",{from:accounts[2]});` `await proveRealCoreInstance.registerMetadata("metadataURI011",{from:accounts[2]});` `await proveRealCoreInstance.registerMetadata("metadataURI012",{from:accounts[2]});` `let md0 = await proveRealCoreInstance.getBrandItemByIndex(0,"Test01234");` `let md1 = await proveRealCoreInstance.getBrandItemByIndex(1,"Test01234");` `let md2 = await proveRealCoreInstance.getBrandItemByIndex(2,"Test01234");` `let md10 = await proveRealCoreInstance.getBrandItemByIndex(0,"Test56789");` `let md11 = await proveRealCoreInstance.getBrandItemByIndex(1,"Test56789");` `let md12 = await proveRealCoreInstance.getBrandItemByIndex(2,"Test56789");` `assert.equal("metadataURI000",md0,"");` `assert.equal("metadataURI001",md1,"");` `assert.equal("metadataURI002",md2,"");` `assert.equal("metadataURI010",md10,"");` `assert.equal("metadataURI011",md11,"");` `assert.equal("metadataURI012",md12,"");` `})`	
Sets up a brand “Test01234” assigned to accounts[1] by calling setBrand using the owner address. Accounts[1] calls the registerMetadata method to register “metadataURI000” Contract Owner address sets the issuance fee to 1 Eth. Accounts[1] issues a certificate with a	`it("Test issuance with correct fee, address and registered metadata", async()=>{` `await proveRealCoreInstance.setBrand(accounts[1],"Test01234");` `await proveRealCoreInstance.registerMetadata("metadataURI000",{from:accounts[1]});` `await proveRealCoreInstance.setIssuanceFee(new BigNumber(web3.utils.toWei("1","ether")));` `let t3 = await proveRealCoreInstance.issue(accounts[2],"metadataURI000",{value:new BigNumber(web3.utils.toWei("1","ether")),from:accounts[1]});` `console.log({'issue gas used' : t3.receipt.gasUsed});` `let issuer = await proveRealCoreInstance.tokenBrands(0);` `assert.equal("Test01234",issuer,"Brand set for certificate correctly");`	

metadata of “metadataURI000” to accounts[2], paying the correct fee. Test successful if the tokenURI and ownerOf items return “metadataURI000” and accounts[2], respectively	```let metadata = await proveRealCoreInstance.tokenURI(0); assert.equal("metadataURI000",metadata,"Metadata set for certificate correctly"); let itemOwner = await proveRealCoreInstance.ownerOf(0); assert(accounts[2],itemOwner,"certificate given to correct Ethereum address") });```	
Sets up a brand “Test01234” assigned to accounts[1] by calling setBrand using the owner address. Accounts[1] calls the registerMetadata method to register “metadataURI000” Contract Owner address sets the issuance fee to 1 Eth. Test successful if accounts[1] attempting to issue while not paying the fee results in reversion	```it("Issuance fails when incorrect fee is used", async() => { await proveRealCoreInstance.setIssuanceFee(new BigNumber(web3.utils.toWei("1","ether"))); await proveRealCoreInstance.setBrand(accounts[1],"Test01234"); await proveRealCoreInstance.registerMetadata("metadataURI000",{from:accounts[1]}); await truffleAssert.reverts(proveRealCoreInstance.issue(accounts[0],"metadataURI000"),"",{from:accounts[1]}); });```	
Sets up a brand “Test01234” assigned to accounts[1] by calling setBrand using the owner address. Accounts[1] calls the registerMetadata method to register “metadataURI000” Contract Owner address sets the issuance fee to 1 Eth. Test successful if accounts[3] (not a registered brand) attempting to issue results in reversion	```it("Issuance fails when non-brand address is used", async() => { await proveRealCoreInstance.setIssuanceFee(new BigNumber(web3.utils.toWei("1","ether"))); await proveRealCoreInstance.setBrand(accounts[1],"Test01234"); await proveRealCoreInstance.registerMetadata("metadataURI000",{from:accounts[1]}); await truffleAssert.reverts(proveRealCoreInstance.issue(accounts[3],"metadataURI000",{from:accounts[2],value:web3.utils.toWei("1","ether")}),""); });```	
Sets up a brand “Test01234” assigned to accounts[1] by calling setBrand using the owner address. Contract Owner address sets the issuance fee to 1 Eth. Test successful if accounts[1] attempting to issue a metadata item they haven’t registered results in reversion	```it("Issuance fails when metadata is not registered", async() => { await proveRealCoreInstance.setIssuanceFee(new BigNumber(web3.utils.toWei("1","ether"))); await proveRealCoreInstance.setBrand(accounts[1],"Test01234"); await truffleAssert.reverts(proveRealCoreInstance.issue(accounts[2],"metadataURI000",{from:accounts[1],value:web3.utils.toWei("1","ether")}),""); });```	
Sets up a brand “Test01234” assigned to accounts[1] by calling setBrand using the owner address. Sets up a brand “Test56789” assigned to accounts[2] by calling setBrand using the owner address. Contract Owner address sets the issuance fee to 1 Eth. Accounts[1] calls the registerMetadata method to register “metadataURI000”	```it("Issuance fails when brand uses metadata they do not own", async() => { await proveRealCoreInstance.setIssuanceFee(new BigNumber(web3.utils.toWei("1","ether"))); await proveRealCoreInstance.setBrand(accounts[1],"Test01234"); await proveRealCoreInstance.setBrand(accounts[2],"Test56789"); await proveRealCoreInstance.registerMetadata("metadataURI000",{from:accounts[1]}); await truffleAssert.reverts(proveRealCoreInstance.issue(accounts[3],"metadataURI000",{from:accounts[2],value:web3.utils.toWei("1","ether")}),""); });```	

Test successful if accounts[2]'s attempt to issue a certificate with metadata of "metadataURI000" results in reversion		
Sets up a brand "Test01234" assigned to accounts[1] by calling setBrand using the owner address. Sets up a brand "Test56789" assigned to accounts[3] by calling setBrand using the owner address. Accounts[1] calls the registerMetadata method to register "metadataURI000" Accounts[3] calls the registerMetadata method to register "metadataURI001" account [1] issues 1 token, account[3] issues 1 token Then account[1] issues a second token Test successful if "test01234" has 2 items in their enumeration and index in the brand enumeration list for the second item account[1] produced corresponds to its id - when queried using getTokenIDFromBrandEnumeration and issuedTokensIndex	it("Brand enumeration is added to correctly after issuance", async() => { await proveRealCoreInstance.setBrand(accounts[1],"Test01234"); await proveRealCoreInstance.setBrand(accounts[3],"Test56789"); await proveRealCoreInstance.registerMetadata("metadataURI000",{from:accounts[1]}); await proveRealCoreInstance.registerMetadata("metadataURI001",{from:accounts[3]}); await proveRealCoreInstance.setIssuanceFee(new BigNumber(web3.utils.toWei("1","ether"))); await proveRealCoreInstance.issue(accounts[2],"metadataURI000",{value:new BigNumber(web3.utils.toWei("1","ether")),from:accounts[1]}); await proveRealCoreInstance.issue(accounts[2],"metadataURI001",{value:new BigNumber(web3.utils.toWei("1","ether")),from:accounts[3]}); await proveRealCoreInstance.issue(accounts[2],"metadataURI000",{value:new BigNumber(web3.utils.toWei("1","ether")),from:accounts[1]}); let noTokens = await proveRealCoreInstance.getNumberOfBrandTokens("Test01234"); assert.equal(noTokens,2,"tokens added to brand enumeration"); let tokenId = await proveRealCoreInstance.getTokenIDFromBrandEnumeration(1,"Test01234"); assert.equal(tokenId,2,"getTokenIDFromBrandEnumeration works correctly"); let tokenIndex = await proveRealCoreInstance.issuedTokensIndex(tokenId); assert.equal(1,tokenIndex,"issuedTokensIndex contains correct data") });	
Setup required for accounts[1] to sucessfully produce a token takes place. Accounts[1] sends two tokens to accounts[2]. Accounts[1] then burns the first of the tokens. Test successful if accounts[1]'s brand enumeration is updated to take into account the first token being removed from the existance.	it("burn correctly removes token from brand enumeration", async() => { await proveRealCoreInstance.setBrand(accounts[1],"Test01234"); await proveRealCoreInstance.registerMetadata("metadataURI000",{from:accounts[1]}); await proveRealCoreInstance.setIssuanceFee(new BigNumber(web3.utils.toWei("1","ether"))); await proveRealCoreInstance.issue(accounts[2],"metadataURI000",{value:new BigNumber(web3.utils.toWei("1","ether")),from:accounts[1]}); await proveRealCoreInstance.issue(accounts[2],"metadataURI000",{value:new BigNumber(web3.utils.toWei("1","ether")),from:accounts[1]}); console.log("issued"); let noTokens = await proveRealCoreInstance.getNumberOfBrandTokens("Test01234"); console.log({"noTokens":noTokens.toString()}); let owner = await proveRealCoreInstance.ownerOf(0); console.log({"owner" : owner}); console.log({"acc 2" : accounts[2]}) await proveRealCoreInstance.burn(0,{from:accounts[2]}); console.log("burned"); noTokens = await proveRealCoreInstance.getNumberOfBrandTokens("Test01234"); assert.equal(noTokens,1,"token deleted from brand enumeration"); });	
Setup required for accounts[1] to sucessfully produce a token takes place. Contract owner's balance is logged Contract owner then calls	it("withdrawBalance allows owner to withdraw balance", async() => { await proveRealCoreInstance.setBrand(accounts[1],"Test01234"); await proveRealCoreInstance.registerMetadata("metadataURI000",{from:accounts[1]}); await proveRealCoreInstance.setIssuanceFee(new BigNumber(web3.utils.toWei("1","ether"))); await	

withdrawBalance to obtain the balance of the contract. The gas cost of this call is also logged. Test sucessful if the contract owner's new balance is equal to their old balance + 1Eth - the withdrawBalance gas cost.	proveRealCoreInstance.issue(accounts[2],"metadataURI000",{value:new BigNumber(web3.utils.toWei("1","ether")),from:accounts[1]}); let initial = await web3.eth.getBalance(accounts[0]); let withdraw = await proveRealCoreInstance.withdrawBalance(); let after = await web3.eth.getBalance(accounts[0]); let cost = new BigNumber(20000000000*withdraw.receipt.gasUsed); initial = new BigNumber(initial); after = new BigNumber(after); console.log({"withdrawBalance gas used" : withdraw.receipt.gasUsed}); assert(after.isEqualTo(initial.plus(new BigNumber(web3.utils.toWei("1","ether"))).minus(cost)), "owner balance after withdrawl is equal to initially + 1eth - cost of calling withdraw"); });	
Setup required for accounts[1] to sucessfully produce a token takes place. Test successful if call by account[3] to steal the contract balance reverts.	it("withdrawBalance prevents non-owners from withdrawing balance", async() => { await proveRealCoreInstance.setBrand(accounts[1],"Test01234"); await proveRealCoreInstance.registerMetadata("metadataURI000",{from:accounts[1]}); await proveRealCoreInstance.setIssuanceFee(new BigNumber(web3.utils.toWei("1","ether"))); await proveRealCoreInstance.issue(accounts[2],"metadataURI000",{value:new BigNumber(web3.utils.toWei("1","ether")),from:accounts[1]}); await truffleAssert.reverts(proveRealCoreInstance.withdrawBalance({from:accounts[3]})); });	

AppendixB: Truffle Test Screenshot

```
kiran@kiran-UX310UAK:~/proverealnft$ truffle test
Using network 'development'.


  Contract: ProveRealCore
{ 'setIssuanceFee gas used': 43118 }
    ✓ setIssuanceFee correctly sets fees when requested by contract owner (72ms)
    ✓ setIssuanceFee prevents non contract owner from changing issuance fee (48ms)
{ 'setBrand gas used': 107783 }
    ✓ setBrand works correctly when called by contract owner (100ms)
    ✓ setBrand prevents non contract owner from setting up a brand (53ms)
    ✓ setBrand prevents contract owner from overwritting a registered brand (82ms)
    ✓ setBrand prevents non contract owners from overwritting a registered brand (76ms)
    ✓ setBrand prevents the registration of a brand to an address already owning a brand (80ms)
{ 'changeBrandAddress gas used': 54326 }
    ✓ changeBrandAddress works correctly when called by contract owner (109ms)
    ✓ changeBrandAddress works correctly when called by brand owner (112ms)
    ✓ changeBrandAddress prevents non contract owner parties to change brand addresses for brands they not own (76ms)
    ✓ changeBrandAddress prevents the changing of a brand address to an address already owning another brand (115ms)
{ 'registerMetadata gas used ': 128521 }
    ✓ registerMetadata adds one new entry correctly (119ms)
    ✓ registerMetadata prevents brands from double entering metadata they already registered (111ms)
    ✓ registerMetadata prevents brands from registering metadata registered by another brand (145ms)
    ✓ getBrandMetadataCount returns the expected value (185ms)
    ✓ isMetadataRegistered returns correct (211ms)
    ✓ getBrandItemByIndex works correctly (438ms)
{ 'issue gas used': 178787 }
    ✓ Test issuance with correct fee, address and registered metadata (237ms)
    ✓ Issuance fails when incorrect fee is used (140ms)
    ✓ Issuance fails when non-brand address is used (137ms)
    ✓ Issuance fails when metadata is not registered (110ms)
    ✓ Issuance fails when brand uses metadata they do not own (257ms)
    ✓ Brand enumeration is added to correctly after issuance (399ms)
issued
{ noTokens: '2' }
{ owner: '0xA3E74e13FA6dcdB5890bCd44FBd1eC8C1649fbd1' }
{ 'acc 2': '0xA3E74e13FA6dcdB5890bCd44FBd1eC8C1649fbd1' }
burned
    ✓ burn correctly removes token from brand enumeration (315ms)
{ 'withdrawBalance gas used': 30064 }
    ✓ withdrawBalance allows owner to withdraw balance (190ms)
    ✓ withdrawBalance prevents non-owners from withdrawing balance (177ms)


  26 passing (7s)
```

AppendixC: Pinata Integration Test Table

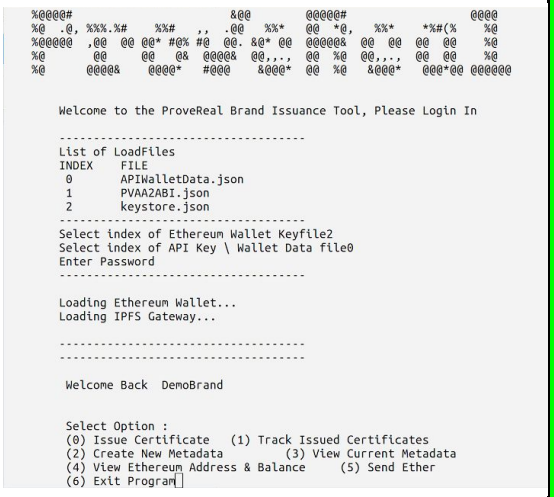
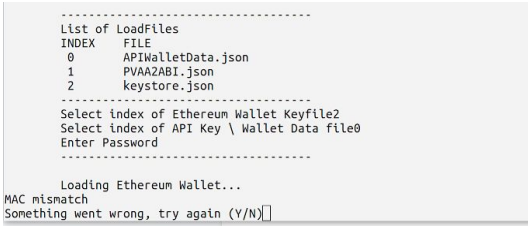
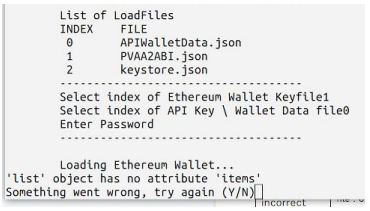
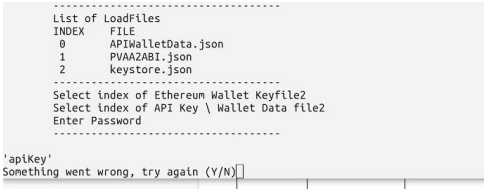
Test Case	Test Description	Test Code	Result
test_init	Verifies if PinataGateway correctly initialised when a legitimate API key is provided	`def test_init(self):` `  a = PinataGateway("b8e2622c756783c562e8","8f4d41219c8a17684825f0d94018225e755eef3eccde5f9202af468f2b8d4e0c")` `  self.assertIsNotNone(a)`	
test_init_bad_keys	Verifies that initialisation of PinataGateway throws an exception when invalid API keys are used	`def test_init_bad_keys(self):` `  with self.assertRaises(Exception):` `PinataGateway("b8e261340134r8","8f4d41219c8a17684825fasfdaagadfa4e0c")`	
test_json	Verifies that the jsonUpload method of PinataGateway conveys the correct data to Pinata. Two dictionaries of format {“name”:””, “description”:””, “image_url”:””} are used as the test data. Randomised string data is used as the the attribute values. This is to prevent regression, due to failure of the method not being caught - due to pre-uploaded data being fetched from IPFS. The test data is uploaded and its presence in pinata is verified by requesting the uploaded data, and checking it matches the input JSON.	`def test_json(self):` `  a = PinataGateway("b8e2622c756783c562e8","8f4d41219c8a17684825f0d94018225e755eef3eccde5f9202af468f2b8d4e0c")` `  randomDictList = []` `  ipfsHashList = []` `  for i in range(2):` `  randomDictList.append({` `"name":''.join([random.choice(string.ascii_letters + string.digits) for n in range(32)]),` `"description":''.join([random.choice(string.ascii_letters + string.digits) for n in range(32)]),` `"image_url":''.join([random.choice(string.ascii_letters + string.digits) for n in range(32)])` `  })` `ipfsHashList.append(a.jsonUpload(randomDictList[i])["IpfsHash"])` `  for i in range(2):` `  start = time.time()` `  r = requests.get("https://gateway.pinata.cloud/ipfs/"+ipfsHashList[i])` `  end  = time.time()` `  print("Fetching JSON ",ipfsHashList[i]," from Pinata took ",end-start," s" )` `  pinataData = r.json()` `  self.assertEqual(randomDictList[i],pinataData)`	
Test_file Upload	Verifies that fileUpload method of PinataGateway conveys the correct data to Pinata. Two text files files consisting of 30K random characters are used as the test data - randomisation is used to prevent regression, due to	`  def test_fileUpload(self):` `  a = PinataGateway("b8e2622c756783c562e8","8f4d41219c8a17684825f0d94018225e755eef3eccde5f9202af468f2b8d4e0c")` `  randomStrList = []` `  ipfsHashList = []` `  for i in range(2):` `randomStrList.append(''.join([random.choice(string.ascii_letters + string.digits) for n in range(30000)]))` `  f = open("test.txt","w")` `  f.write(randomStrList[i])` `  f.close()` `  ipfsHashList.append(a.fileUpload("test.txt")["IpfsHash"])`	

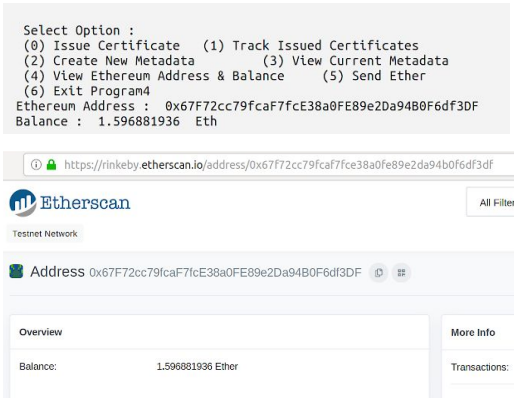
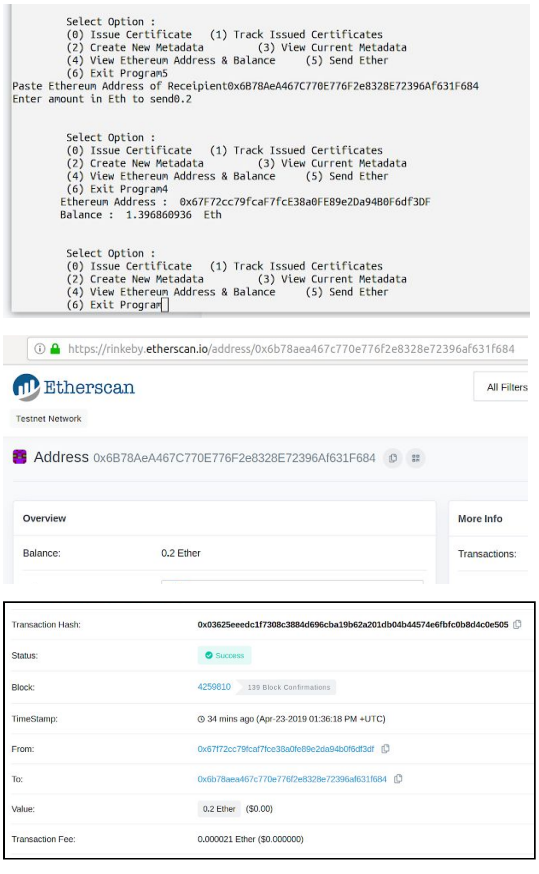
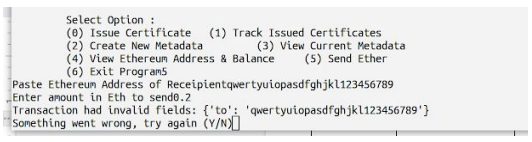
	failure of the method not being caught - due to pre-uploaded data being fetched from IPFS. Using large text files serves as a good analogue for the image files that IssuanceTool.py will be used to upload The test data is uploaded and its presence in pinata is verified by requesting the uploaded data, and checking it matches the input files..	`for i in range(2):` `start = time.time()` `r = requests.get("https://gateway.pinata.cloud/ipfs/"+ipfsHashList[i])` `end = time.time()` `print("Fetching string ",ipfsHashList[i]," from Pinata took ",end-start," s" )` `pinataData = r.text` `self.assertEqual(randomStrList[i],pinataData)`	
test_bad_str_json	Verifies jsonUpload throws an exception when a str is given instead of a dict	`def test_bad_str_json(self):` `a = PinataGateway("b8e2622c756783c562e8","8f4d41219c8a17684825f0d94018225e755eef3eccde5f9202af468f2b8d4e0c")` `randomDictList = []` `with self.assertRaises(Exception):` `a.jsonUpload("nonsenseString")`	
test_json_ipfsio	Works using same logic as test_json, but instead requests the IPFS gateway managed by IPFS.io for the data - with a 30 second timeout.	`def test_json_ipfsio(self):` `a = PinataGateway("b8e2622c756783c562e8","8f4d41219c8a17684825f0d94018225e755eef3eccde5f9202af468f2b8d4e0c")` `randomDictList = []` `ipfsHashList = []` `for i in range(2):` `randomDictList.append({` `"name":''.join([random.choice(string.ascii_letters + string.digits) for n in range(32)]),` `"description":''.join([random.choice(string.ascii_letters + string.digits) for n in range(32)]),` `"image_url":''.join([random.choice(string.ascii_letters + string.digits) for n in range(32)])` `})` `ipfsHashList.append(a.jsonUpload(randomDictList[i])["IpfsHash"])` `for i in range(2):` `r = requests.get("https://ipfs.io/ipfs/"+ipfsHashList[i],timeout=30)` `end = time.time()` `print("Fetching JSON ",ipfsHashList[i]," from ipfs.io took ",end-start," s" )` `ipfsData = r.json()` `self.assertEqual(randomDictList[i],ipfsData)`	
Test_fileUpload_ipfsio	Works using same logic as test_fileUpload, but instead requests the IPFS gateway managed by IPFS.io for the data - with a 30 second timeout.	`def test_fileUpload_ipfsio(self):` `a = PinataGateway("b8e2622c756783c562e8","8f4d41219c8a17684825f0d94018225e755eef3eccde5f9202af468f2b8d4e0c")` `randomStrList = []` `ipfsHashList = []` `for i in range(2):` `randomStrList.append(''.join([random.choice(string.ascii_letters + string.digits) for n in range(30000)]))` `f = open("test.txt","w")` `f.write(randomStrList[i])` `f.close()` `ipfsHashList.append(a.fileUpload("test.txt")["IpfsHash"])` `for i in range(2):` `start = time.time()` `r = requests.get("https://ipfs.io/ipfs/"+ipfsHashList[i],timeout=90)` `end = time.time()` `print("Fetching string ",ipfsHashList[i]," from ipfs.io took ",end-start," s" )` `ipfsData = r.text` `self.assertEqual(randomStrList[i],pinataData)`	

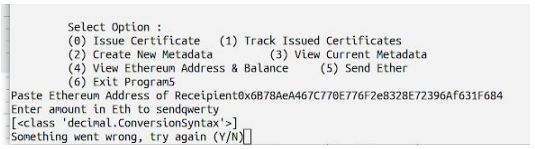
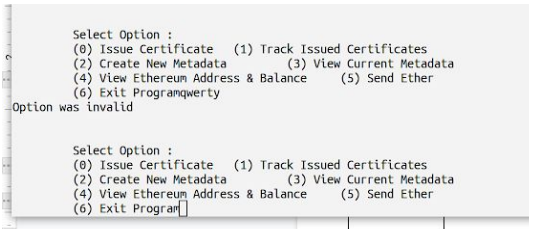
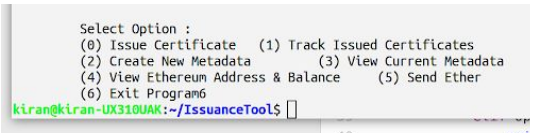
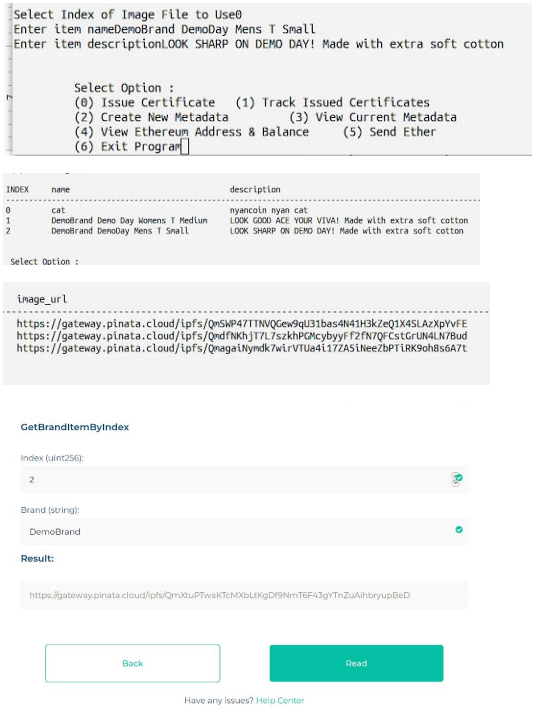
AppendixD:Pinata Integration Test Screenshot

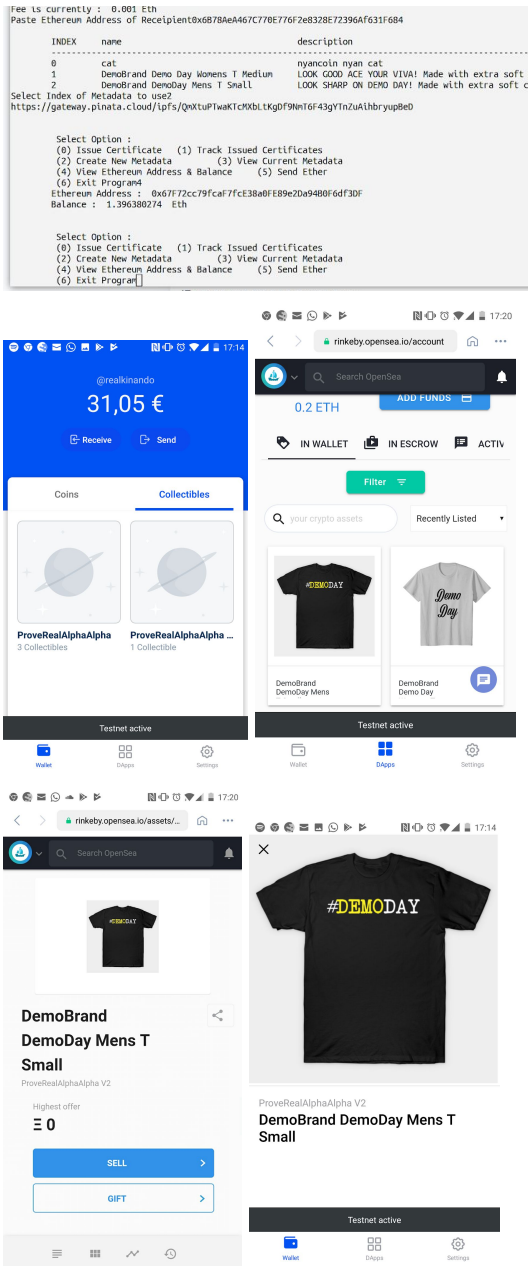
```
kiran@kiran-UX310UAK:~/IssuanceTool$ python3 PinataGateway_test.py
.Fetching string  QmQuQtoyMvDJqimW9joSH4rxTr8L1FXCHbXpV4Vg2EtCxu  from Pinata took  0.20387673377990723  s
Fetching string  QmSYsMtoBadL1RiArhJGTTbwcrsVQdsf8snHnpb22GSvgU  from Pinata took  0.19736766815185547  s
.E..Fetching JSON  QmectpKryeo9ju8xprGMxhKH9r7Jj2F4YYa1P5qZXgGKn2  from Pinata took  0.16150283813476562  s
Fetching JSON  QmQ2rX7y9KEuYAJx1YDfv3gb7xSLavHCTRsaKRQC1rrn4f  from Pinata took  0.1421804428100586  s
.Fetching JSON  QmW4wFssDELRR69wm82iUxVr8Zr3NZqFzVSPYrnXGGAN9W  from ipfs.io took  0.22023916244506836  s
Fetching JSON  QmQRnWfFsu88tEevNWzitrCRnt5vepSdB1RNVXZQv4VNuo  from ipfs.io took  0.18443012237548828  s
.
======================================================================
ERROR: test_fileUpload_ipfsio (__main__.TestPinataGateway)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/home/kiran/.local/lib/python3.6/site-packages/urllib3/connectionpool.py", line 384, in _make_request
    six.raise_from(e, None)
  File "<string>", line 2, in raise_from
  File "/home/kiran/.local/lib/python3.6/site-packages/urllib3/connectionpool.py", line 380, in _make_request
    httplib_response = conn.getresponse()
  File "/usr/lib/python3.6/http/client.py", line 1331, in getresponse
    response.begin()
  File "/usr/lib/python3.6/http/client.py", line 297, in begin
    version, status, reason = self._read_status()
  File "/usr/lib/python3.6/http/client.py", line 258, in _read_status
    line = str(self.fp.readline(_MAXLINE + 1), "iso-8859-1")
  File "/usr/lib/python3.6/socket.py", line 586, in readinto
    return self._sock.recv_into(b)
  File "/usr/lib/python3.6/ssl.py", line 1012, in recv_into
    return self.read(nbytes, buffer)
  File "/usr/lib/python3.6/ssl.py", line 874, in read
    return self._sslobj.read(len, buffer)
  File "/usr/lib/python3.6/ssl.py", line 631, in read
    v = self._sslobj.read(len, buffer)
socket.timeout: The read operation timed out

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "/home/kiran/.local/lib/python3.6/site-packages/requests/adapters.py", line 449, in send
    timeout=timeout
  File "/home/kiran/.local/lib/python3.6/site-packages/urllib3/connectionpool.py", line 638, in urlopen
    _stacktrace=sys.exc_info()[2])
  File "/home/kiran/.local/lib/python3.6/site-packages/urllib3/util/retry.py", line 368, in increment
    raise six.reraise(type(error), error, _stacktrace)
  File "/home/kiran/.local/lib/python3.6/site-packages/urllib3/packages/six.py", line 686, in reraise
    raise value
  File "/home/kiran/.local/lib/python3.6/site-packages/urllib3/connectionpool.py", line 600, in urlopen
    chunked=chunked)
  File "/home/kiran/.local/lib/python3.6/site-packages/urllib3/connectionpool.py", line 386, in _make_request
    self._raise_timeout(err=e, url=url, timeout_value=read_timeout)
  File "/home/kiran/.local/lib/python3.6/site-packages/urllib3/connectionpool.py", line 306, in _raise_timeout
    raise ReadTimeoutError(self, url, "Read timed out. (read timeout=%s)" % timeout_value)
urllib3.exceptions.ReadTimeoutError: HTTPSConnectionPool(host='ipfs.io', port=443): Read timed out. (read timeout=90)

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "PinataGateway_test.py", line 91, in test_fileUpload_ipfsio
    r = requests.get("https://ipfs.io/ipfs/"+ipfsHashList[i],timeout=90)
  File "/home/kiran/.local/lib/python3.6/site-packages/requests/api.py", line 75, in get
    return request('get', url, params=params, **kwargs)
  File "/home/kiran/.local/lib/python3.6/site-packages/requests/api.py", line 60, in request
    return session.request(method=method, url=url, **kwargs)
  File "/home/kiran/.local/lib/python3.6/site-packages/requests/sessions.py", line 533, in request
    resp = self.send(prep, **send_kwargs)
  File "/home/kiran/.local/lib/python3.6/site-packages/requests/sessions.py", line 646, in send
    r = adapter.send(request, **kwargs)
  File "/home/kiran/.local/lib/python3.6/site-packages/requests/adapters.py", line 529, in send
    raise ReadTimeout(e, request=request)
requests.exceptions.ReadTimeout: HTTPSConnectionPool(host='ipfs.io', port=443): Read timed out. (read timeout=90)

----------------------------------------------------------------------
Ran 7 tests in 103.984s

FAILED (errors=1)
kiran@kiran-UX310UAK:~/IssuanceTool$
```

AppendixE: System Test Table

Test Case	Test Data	Expected Result	Test Results	Success
Login allowed when password, keystore and apiKeyfile are all correct	Ethereum Wallet Keyfile : 2 API Key \ Wallet Data file : 0 Password : TestPass01234TestPass	Menu screen loads, Message “Welcome back DemoBrand”		
Login not allowed when password is incorrect	Ethereum Wallet Keyfile : 2 API Key \ Wallet Data file : 0 Password : 01234	Prompt : “something went wrong, do you want to try again”		
Login not allowed when keyfile is incorrect	Ethereum Wallet Keyfile : 1 API Key \ Wallet Data file : 0 Password : TestPass01234TestPass	Prompt : “something went wrong, do you want to try again”		
Login not allowed when api/wallet data file is incorrect	Ethereum Wallet Keyfile : 2 API Key \ Wallet Data file : 2 Password : TestPass01234TestPass	Prompt : “something went wrong, try again”		

When loggged in user can view address and balance	Option : 4	Address: 0x67F72cc79fcaF7fcE38a0FE89e2Da94B0F6df3DF Balance matching that found for the address using Etherscan		
When logged in users can view previously issued certificates	Option : 1	Table containing the three previously issued certificates - with the correct values in the correct format		
Send Ether works correctly when correct inputs are used	Option : 5 Address : 0x6b78aea467c770e776f2e8328e72396af631f684 Eth : 0.2	Transaction goes through, balance shown within the issuance tool has been updated to reflect the new state of the blockchain. Balance of receiver goes from 0 to 0.2 eth		
Send Ether prevents transaction when address is invalid	Option : 5 Address : qwertyuiopasdfghjkl123456789 Eth : 0.2	Prompt : “something went wrong, try again”		

format				
Send Ether prevents transaction when eth value is invalid	Option : 5 Address : 0x6b78aea467c770e776f2e8328e72396af631f684 Eth : qwerty	Prompt : “something went wrong, try again”		
Menu correctly handles invalid option input	Option : qwerty	Prompt : “option was invalid” Menu reappears		
Program correctly closes when exit option is given	option:6	Program terminates		
User can register metadata correctly	Option:2 ImageFIle Index : 0 (demoDay2.jpg)  Name: DemoBrand DemoDay Mens T Small Description : LOOK SHARP ON DEMO DAY! Made with extra soft cotton	IssuanceTool.py uses pinataGateway.py to upload an image file to IPFS, uses the returned hash to form a JSON file for the metadata. This can be verified by viewing the metadataURI submitted to the blockchain and confirming it contains the correct data. registerMetadata command gets sent to the blockchain. This is verifiable by viewing the updated metadata list within the app and the confirming the transaction took place using etherscan and myetherwallet.	 https://gateway.pinata.cloud/ipfs/QmXtuPTwaKTcMXbLtKgDf9NmT6F43gYTnZuAihbryupBeD https://gateway.pinata.cloud/ipfs/QmagaiNymdk7wirVTUa4i17ZA5iNeeZbPTiRK9oh8s6A7t	

Issuance Works when valid address is entered	Option:0 Address : 0x6b78aea467c770e776f2e8328e72396af631f684 Metadata Index : 2	IssuanceTool should call the contract's issue method using the PVAAGateway, to issue the certificate to the user (coinbase wallet user) This can be verified by opening the Coinbase Wallet Dapp to confirm the transaction has occurred, and the certificate has been created and issued with the correct metadata.		
Issuance Tool correctly Handles entry of incorrect address	Option:0 Address : qwerty Metadata Index : 2	Prompt : "something went wrong, try again"		

Appendix F: Gas Cost Table

Transaction Type	Gas Cost in	Gas Cost in	Rinkeby Gas /	Rinkeby Tx

	unittest	Rinkeby	Truffle Gas	Hash
Issue	178787	302340	1.6911	0xd14656f86cb3b739dc59ed3ba9ca24e2e069c47762da22df0dd77eca30853616
RegisterMetadata	128521	178322	1.3875	0xd72420504d78b313376bfc484dae41e8aad8297f2607227fb2085c5738acb1f7
SetBrand	107783	107911	1.0012	0x3c568317dc8e01309f1db5f286e69761cf42a610dd48ed832c094dea65ec3ce3
ChangeBrandAddress	54326	54033	0.9946	0xff92a103f32ca172b08b5a3dba4348cca4fe99d2cb6f05d235a47fe85774336b
SetIssuanceFee	43118	28118	0.6521	0xe943d8a8e77830535757da77e78c1e6011644a6bfff5ed273251552083390bfd
WithdrawBalance	30064	30064	1	0xcd66e70d64a7bdbef4b6deac24efbee4c568cf881ceff9d67a180f2d32eb7419

Appendix G: Gas Ratio Box Plot dataset

Analysis of Rinkeby Gas / Truffle Gas Values for dataset in Appendix F

	Value
Max	1.6911
Min	0.6521
Median	1.0006
Mean	1.1211
Q3	1.3875

Q1	0.9946